Hardware Acceleration of Machine Learning

Evaluation and comparison of different hardware-aware optimization techniques

Master's thesis in Computer Science and Engineering

Fangzhou Chen
William Sköld

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2023

Master's thesis 2023

Hardware Acceleration of Machine Learning

Evaluation and comparison of different hardware-aware optimization techniques

Fangzhou Chen
William Sköld

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2023

Hardware Acceleration of Machine Learning
Evaluation and comparison of different hardware-aware optimization techniques
Fangzhou Chen, William Sköld

Supervisor: Pedro Petersen Moura Trancoso, Department of Computer Science and Engineering
Advisor: Evangelos Siminos, Volvo Group (SML)
Examiner: Pedro Petersen Moura Trancoso, Department of Computer Science and Engineering

Master's Thesis 2023
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: A running megatron, generated by DALLE2.

Typeset in LaTeX
Gothenburg, Sweden 2023

Hardware Acceleration of Machine Learning
Fangzhou Chen
William Skold
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

The Transformer architecture has been widely used in various fields, as demonstrated by GPT-3, a large language model that shows impressive performance. However, achieving such excellent performance requires high computational capabilities. Therefore, improving the computational power of current machine learning systems is of great importance.

This thesis aims to optimize and accelerate fine-tuning of Transformer-based models while taking into account several evaluation criteria, such as training time, energy consumption, cost, and hardware utilization. Additionally, a comparison is made between GPU training settings and specialized AI accelerators, such as TPU training settings.

In our study, a high-performance kernel for the Adan optimizer was introduced, and the LightSeq library is applied to accelerate existing Transformer components. We also introduce mixed precision training into our workflow and compare all these optimization techniques step by step with baseline performance. In addition, our analysis includes distributed training with multiple GPUs, and a backpropagation time estimation algorithm is introduced. Next, Google's TPU accelerator is used to run our task, and its performance is compared to the similar GPU setup used in our study. Finally, the advantages and disadvantages of different methods are systematically analyzed, while training on V100, A100, A10 and T4 with different configurations. Meanwhile, the workflow between GPUs and TPUs is analyzed, illustrating the pros and cons of different accelerators.

Various weights for measuring optimization methods based on time, energy consumption, cost, and hardware utilization are proposed. Our analysis shows that optimal scores in all metrics can be achieved by implementing the optimized LightSeq model, kernel fusion for the Adan optimizer, and enabling mixed precision training. While training with TPU offers certain advantages, such as large batch sizes when loading training data, the ease of use, reliability, and software stability of GPU training surpasses that of TPU training.

Keywords: Transformer, GPU, Distributed, Energy consumption, Fine-tuning, TPU.

Acknowledgements

We would like to express our sincere gratitude to our supervisors, Evangelos and Pedro for their guidance, expertise and unwavering support throughout this thesis. Their mentorship, advice and constructive feedback have been instrumental in shaping the direction and results of this thesis. We would also want to thank Thomas Nordenskjöld, at Volvo for his support and help with all the small things related to collaborating with Volvo for this thesis.

Lastly, we'd like to extend our thanks to Volvo Group SML for providing us with the necessary tools for this thesis, such as Microsoft Azure for benchmarking, expertise from employees and workspace.

William and Fangzhou, Gothenburg, 2023-06-13

Contents

List of Figures

List of Tables

1

Introduction

Machine learning (ML) has become increasingly popular in recent years as a powerful tool for solving complex tasks in natural language processing (NLP), computer vision, and automation, etc. The emergence of impressive ML technologies such as ChatGPT [1] and DALLE [2], which showcased remarkable capabilities in early 2023, is due to developments in deep learning, a branch of ML. These advancements are based on the Transformer architecture proposed by Google [3].

The Transformer-based BERT model [4] is extensively utilized in the field of NLP. Efficiency gains can be observed by a wide range of other industries and companies. As outlined in a blog post by Google showing utilize BERT enhance the interpretation of user queries [5] to improve user experience. Examples such as the ones listed show that even for domain specific tasks, using the likes of transformer-based models can improve efficiency and performance. Some AI-based translation tools, such as DeepL, also use Transformer-based models. By customizing the network architecture to a certain extent, translation accuracy and speed between different languages are greatly improved, while maintaining low usage costs.

The success of the Transformer architecture can be attributed to its ability to address the data dependency problem that arises in training recurrent neural network (RNN) commonly used in traditional NLP, which accelerates the transformer-based model's training through parallelization [6]. In the training of RNN, the input of the next iteration depends on the output of the previous round, making it difficult to feed multiple tokens in the sequence into the network simultaneously for parallel training. However, Transformer eliminates this dependency by introducing the self-attention mechanism, which replaces the association between tokens with the self attention mechanism.

However, the Transformer architecture is not without its flaws, one of which is its lower computational efficiency. Specifically, if the input token sequence has a length of n and each token embedding has dimension d, the overall complexity per layer of the self-attention mechanism is $O(n^2d)$. In comparison, the complexity per layer of CNN in the same case is $O(knd^2)$ (k is the number of convolution kernels) and RNN is $O(nd^2)$. This shows that the complexity of Transformer increases quadratically as the sequence length grows linearly. Therefore, it is essential to improve the compute capabilities for Transformer architecture from both hardware side and software side, especially when dealing with longer sequences.

1.1 Machine Learning Hardware

There are several companies offering various hardware prototypes that can accelerate the training of machine learning models. NVIDIA's GPU is the most commonly used for accelerating CUDA-based applications like machine learning, thanks to its high performance and built-in CUDA core. Other companies like Intel and AMD also offer their own selection of GPUs suitable for accelerating deep learning. For example, Intel's latest ARC series GPUs are expected to perform similarly to NVIDIA's RTX series. Intel has implemented an abstraction of its latest GPU offering through its XPU heterogeneous computing platform, enabling it to run on mainstream machine learning frameworks like PyTorch. Similarly, AMD offers the Radeon family of GPUs suitable for ML using its open-source software called ROCm.

In addition to GPUs, other types of hardware can accelerate machine learning, including FPGAs or specialized chips designed specifically for ML. For example, the Cerebras Wafer-Scale Engine (WSE) chip is a giant AI accelerator that utilizes a single wafer. It allows for reduced communication latency between cores and more shared on-chip memory, resulting in highly accelerated training and ultra-low latency inference [7]. Another example is that of Google's Tensor Processing Unit (TPU), which will be covered in more details later on.

1.2 Machine Learning Framework

Data scientists usually use a comprehensive deep learning framework to aid them in creating deep learning models. The most common frameworks in academia and industry are PyTorch [8] and Tensorflow [9]. The primary component of these frameworks is a data structure known as a Tensor. Torch providing exceptional support for GPUs, but it only offers an upper-level interface to the programming language Lua, which is not widely used. As a result, Facebook's development team reprogrammed the upper-level interface in Python to provide full Python support, which now named PyTorch. PyTorch has gained popularity because of the widespread use of Python in the community.

TensorFlow is a deep learning framework introduced by Google in 2015. It is comparable to PyTorch since operations are performed on tensors. However, unlike PyTorch, TensorFlow is a static graph-based framework, whereas PyTorch is based on dynamic graphs. Both dynamic and static graphs have their advantages and disadvantages. Dynamic graphs are simple to use for model development and error detection. Nonetheless, they often have lower performance and cannot meet specific needs. Static graphs are quicker since they are designed for a series of optimizations to be made before execution. However, the optimized graphs lead to harder to detect errors during runtime making debugging and error solving more difficult.

1.2.1 Machine Learning Libraries

In addition to hardware, high-performance software libraries are also important in achieving maximum computational performance. Different hardware requires

different libraries to fully exploit their potential performance. For example, NVIDIA provides the cuDNN high-performance machine learning software library, while Intel offers the OneDNN machine learning library. AMD provides the ROCm library for their hardware to maximize performance in machine learning tasks. These libraries use a variety of hardware optimization strategies to address performance bottlenecks that compilers cannot optimize.

For instance, the double buffering technique proposed by *Tian* et al. [10] takes advantage of the fast warp switching feature of NVIDIA GPUs. This technique hides memory access latency on the previous warp by prefetching data for the other warp, allowing the scheduler to quickly switch the other warp to the idle stream processor when a memory access occurs on one warp. However, the increasing diversity of hardware in recent years has presented a significant challenge in the development of software libraries. While specific hardware designs can significantly improve hardware performance in a given domain, non-specialists may struggle to accelerate their tasks by writing high-performance kernels independently, as these efforts are usually limited to hardware vendors.

1.3 Transformer usage at Volvo

In international companies like Volvo Group, many teams are trying to use artificial intelligence technology to expedite their business and reduce costs. The implementation of Transformer-based neural network architecture is being pursued by many Volvo data scientists to build their own large language model and train a customized model on the internal corpus. This has sparked Volvo's interest in speeding up training the Transformer model, which would allow AI technology to use in Volvo's various business activities by reducing the cost of training large language models.

At Volvo SML, BERT-based models are used in various tasks, such as predicting traffic codes by generating corresponding codes based on parts' descriptions. This reduces the need for human labor and allows human effort to be focused on more critical steps. Our goal is to optimize the Transformer architecture to reduce hardware usage and time costs associated with the model development workflow.

1.4 Problem statement

Nowadays, the push for training a large model within academic researchers and industry engineers has increased, pre-trained models based on the Transformer architecture such as BERT are already prevalent in NLP-based tasks. However, they still require fine-tuning on domain specific datasets before they can effectively be in research or products. The fine-tuning process incurs both an initial cost but also a recurring cost in terms of monetary costs, time and energy, due to factors such as wanting to improve the model in the future with more data, or simply model degradation in which the models performance worsens with time. This thesis therefore aims to investigate the effectiveness of various hardware-aware optimization techniques such as kernel fusion, mixed precision, acceleration libraries and distributed

training techniques when fine-tuning a pre-trained BERT model. The results will address the following research questions:

1. How can the fine-tuning process of the BERT model benefit from hardware-aware optimization techniques in terms of time, energy consumption and monetary cost?
2. What are the benefits and drawbacks of using GPU-based training methods as opposed to a TPU-based training method?

1.5 Related Work

Sparsh et al. [11] presents a survey focusing on the architecture and system-level optimizations on GPUs for DL applications. Reviewing techniques for inference and training as well as for using either single GPU or a distributed system. Highlighting that over the past nine years, GPU compute power has increased 32x, whilst the memory bandwidth only increased 13x whilst the DNN memory requirements grow at a rapid pace. Thus, coming to the conclusion that substantial improvements within memory capacity and bandwidth for GPUs are required to keep GPUs the main platform for DNNs. Most papers focused only on a single technique for increasing performance, however future work should use multiple techniques in conjunction and evaluate them.

Further more, they state that a critical challenge in GPU use is large power consumption, and that nearly all papers surveyed ignored the energy efficiency assessment of their solutions that future workers should evaluate. As well as newer GPU models and their low / mixed-precision techniques [12] for achieving higher efficiency. Moreover, they say it will be interesting to see how next-generation GPUs compare to custom-made AI accelerators such as TPUs. Finally, they focus on the lack of security when using GPUs, and that security needs to be a design parameter when designing GPUs and CNNs, not retrofitted.

Shi et al. [13]present a comparative study of SOTA GPU-accelerated DL software tools (Caffe, TensorFlow, Torch, etc.), benchmarking tool performance using 3 popular neural networks on two CPU and three GPU platforms. They also benchmark on a distributed system with multiple GPUs. Two contributions are made, one for end-users and one for software developers. First, by using their results as a guide for selecting hardware and software by end-users. Secondly, by showing software developers of DL tools possible future directions for performance optimization.

They conclude by showing that the tools tested make good use of GPUs, and achieve significant speed-ups over CPU counterparts. However, there is no single software tool evaluated that can consistently perform better than another, so there should be opportunities for optimization in this area. Finally, they stated that there were two directions for future work. One is benchmarking DL software tools and hardware platforms, and the other is evaluating tool scalability on a high-performance GPU cluster.

Sze et al. [14] discuss the challenges of energy consumption, throughput, and data

movement and how to address them at various levels of hardware design. They state that data movement is the cause of high energy consumption, and that recent research focuses on reducing memory movement while maintaining accuracy, throughput, and cost. They therefore advocate selecting architectures with efficient memory hierarchies and data flows that increase data reuse.

They also mention that joint design between algorithms and hardware has been a driving force to reduce data movement requirements through reduced bit width precision, compression, and increased sparsity. They conclude that considering the interactions between different levels of hardware design, one example is reducing the cost of memory access, which could lead to more energy-efficient data flows.

Wang et al. [15] conduct a study on the performance and energy efficiency of multiple hardware types (Intel CPU, NVIDIA GPU, AMD GPU, Google TPU) on a suite of representative DL workloads. They investigate the impact on performance and energy consumption based on many factors, such as hardware, software library for hardware, and deep learning framework. Their contribution is twofold, providing end users with useful results when selecting hardware for their own projects, and exposing opportunities for hardware manufacturers to improve their software.

1.6 Aim and Objectives

The main aim is to analyze how hardware-aware optimization techniques such as Kernel Fusion, Multi-tensor Access, etc impacts the fine-tuning process when fine-tuning on the BERT Model. Secondarily, we also aim to investigate the performance differences between various hardware within two main architectures, GPUs and TPUs as well as compare these architectures directly. Based on these aims, the following goals are formulated:

1. Accelerating the Adan optimizer using Kernel fusion and Multi-tensor access techniques. Different hardware-related optimization techniques will be explored to determine the best implementation. The advantages of the high-performance kernel in terms of computation time and hardware utilization will be evaluated.

2. Accelerating the existing training process (Hugging Face baseline) using techniques such as mixed precision, distributed training and also using acceleration libraries such as LightSeq. The optimization process will be conducted in stages, and compared with the existing workflow, mainly evaluating the training time, energy consumption, and cost.

3. The existing workflow will be migrated to TPU-based training platform and compared with the GPU-based training platform, both in terms of baseline performance as well as optimized performance. The advantages and disadvantages of both methods will be analyzed.

4. Lastly, performing a systematic benchmark / grid search to evaluate a plethora of hardware, and how the hardware is impacted by the optimization techniques used in this thesis.

1.7 Delimitations

This thesis has one main limitation, access to physical hardware. Because of this, the thesis will be conducted in the cloud only, that is, on either Microsoft Azure or Google Cloud. The advantage of this is access to a wide selection of hardware that can be benchmarked, although at the disadvantage of having to use energy estimations rather than real energy measurements.

However, even though a wide choice of hardware is available in the commercial market, our thesis is limited to the use of NVIDIA GPUs available in Volvo's subscription on Azure and Google TPUs (v2 and v3). The TPUv4 would perhaps be a more fitting candidate for some of the benchmarks but is unavailable due to monetary reasons (Google Cloud offers TPUv2 and TPUv3 for free via their TPU Reserach Cloud initiative). Moreover, while pre-training would be an interesting area of research, the costs associated with repeating pre-training using cloud computing are considered too high.

1.8 Thesis Outline

Theory: In this chapter, we provide a brief overview of relevant background literature, including the Transformer and its key algorithms, optimization algorithms, GPGPU and TPU hardware architectures, LightSeq library, mixed precision training, and distributed training.

Method: This chapter outlines the current problems in the training process and presents different optimization methods for each problem to improve training performance. We also discuss the potential benefits of these optimization methods.

Results: In this chapter, we present the results of our approach and briefly describe them.

Discussion: This chapter synthesizes the results from the previous chapter, summarizes our findings, and provides recommendations. We also delve deeper into the optimization methods and training processes used to provide insights useful for future work.

Future work: In this chapter, we suggest future work based on our findings as well as based on the limitations of the thesis.

Conclusion: This chapter summarizes the thesis content and presents our conclusion based on the presented results.

2

Theory

This chapter serves as an introduction to the underlying theory that forms the basis of this thesis. It will cover several key topics, including the Transformer architecture, deep learning optimizer theory, GPU and TPU architectures, the LightSeq software library, mixed precision training, and the concept of distributed training. By providing an overview of these theories, the reader will gain a better understanding of the subsequent chapters of the thesis and how these theories relate to the research presented in this thesis.

2.1 Transformer

Transformer is a popular machine learning architecture used for various tasks in natural language processing. These tasks include sequence labeling, sequence classification, natural language generation, word sense disambiguation, etc. The Transformer architecture, with its superior semantic feature extraction and extended sequence processing capabilities, has enabled Transformer-based models to achieve state-of-the-art (SOTA) performance on numerous datasets. ERNIE-Doc-Large [16] achieved SOTA performance on the IMDB dataset text classification task with 97.1% accuracy. On the Penn Treebank dataset [17] GPT-3 (Zero shot) achieved the SOTA test perplexity on language modelling of 20.5 [18].

This thesis focuses on sequence classification tasks using the Transformer-based BERT model. The Transformer architecture introduces a range of novel features and capabilities which will illustrated in the following sub-sections.

2.1.1 Self Attention

Self-attention is a technique that allows the aggregation of all positions in an input sequence, weighing each position according to its importance. This approach significantly enhances a model's ability to capture long-range dependencies within a sequence. The Transformer architecture uses this technique to achieve better performance than traditional sequence models such as recurrent neural networks (RNNs) [19] and long short-term memory (LSTM) [20] networks when dealing with long sequences. Moreover, the Transformer architecture does not rely on sequential computations over time, as opposed to RNNs and LSTMs. This feature allows efficient parallel computation during training, resulting in faster processing times. This ad-

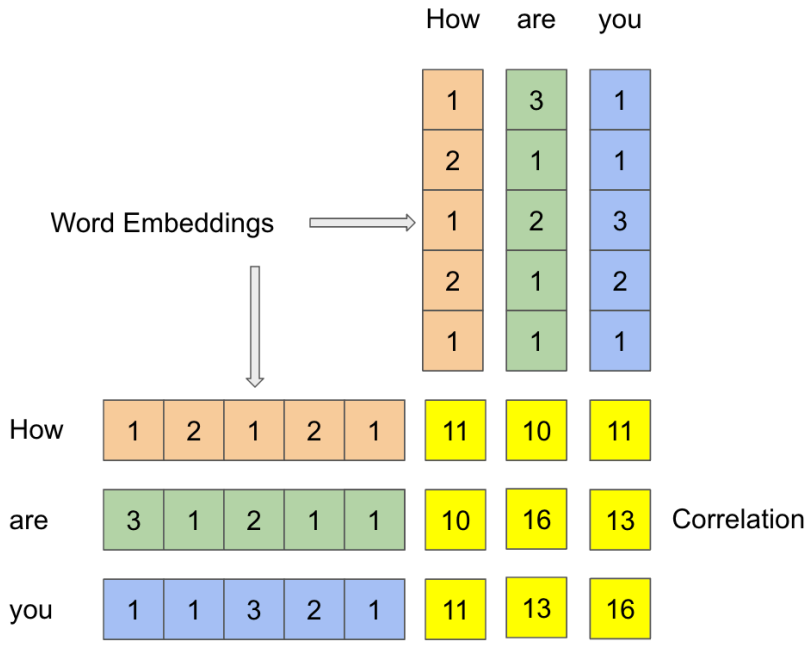

Figure 2.1: Example of correlation between query vector and key vector.

vantage over traditional sequence models can significantly reduce the time required for training.

There are three essential parameters for the attention mechanism, the Query, Key and Value matrices (Q, K, V) as depicted in the following equation,

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V. \tag{2.1}$$

Notably, the Query, Key, and Value matrices are derived from the word embedding vectors through linear transformations. As illustrated,

$$\begin{aligned} \text{Embedding} \times W^Q &= Q \\ \text{Embedding} \times W^K &= K \\ \text{Embedding} \times W^V &= V \end{aligned} \tag{2.2}$$

where W^Q, W^K, W^V are trainable parameters.

Q matrix multiplied by the transpose of the K matrix produces the correlation matrix between the Query and Key. A higher correlation between Query and Key results in a larger corresponding value. This procedure is important because it allows the strength of the relationship between Query and Key to be determined in a structured and quantitative manner, allowing us to enhance the original word embedding.

The correlation computation is shown in Figure 2.1, in order to simplify the expression the original embeddings are used directly. This means that the Query matrix on the left and the Key matrix on the top have the same values, and both matrices undergo a linear transformation during actual computation. The yellow correlation matrix reflects the correlation between each token, with higher values indicating a stronger correlation with the current token, and therefore requiring a higher weight. These weights are then embedded as contextual information from the corpus into the original vector.

Normalization of dimension is necessary to obtain reasonable values for relationships, and to maintain stability in the gradients, which is crucial for neural networks. Subsequently, the correlation matrix is multiplied by the Value matrix to obtain the attention-weighted word embedding representation.

Attention mechanism aims to improve the learning process by identifying relevant tokens and strengthening their weights. This is accomplished by comparing associations between sequence contexts and amplifying weights, resulting in an enhanced representation of the original word embeddings in the current context.

2.1.2 Multi-headed Attention

Multi-head attention mechanism utilizes multiple sets of trainable matrices, namely W_i^Q, W_i^K, and W_i^V, to project input data into different subspaces. By initializing these matrices randomly, each head captures distinct semantic information, leading to diverse feature representations. Subsequently, the output features from different heads are concatenated and transformed by W_O, which is the linear layer, to obtain the final embedding representation, as seen here:

$$\begin{aligned} \text{MultiHead}(Q,K,V) &= \text{Concat}(\text{head}_1, \text{head}_2, ..., \text{head}_h)W^O \\ \text{head}_i &= \text{Attention}(QW_i^Q, KW_i^K, VW_i^V). \end{aligned} \tag{2.3}$$

This approach facilitates the learning of distinct pieces of information in various representation spaces, which enhances the flexibility and efficiency of the model. The design is reminiscent of convolution neural networks, which employ a large number of convolution kernels to capture features in feature space by randomly initializing the values of different kernels. Increasing the number of heads in the Transformer model can enhance its ability to capture sequence features.

Figure 2.2 shows that the multi-head attention mechanism exhibits a high degree of parallelism because the attention heads do not have interconnections and can therefore be computed in parallel. This property provides the model with increased flexibility and perceptual capabilities, while making efficient use of computational resources without encountering training obstacles.

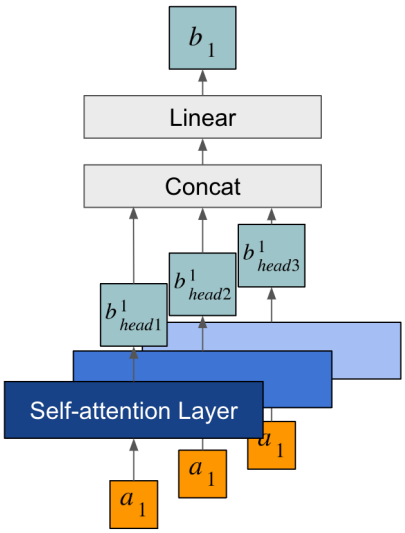

Figure 2.2: Multi Attention with three attention heads, vector a_1 is word embedding and b^1_{headi} is the attention vector generated by different attention heads, and these vectors were transformed by the linear layer.

2.1.3 Transformer Architecture

Transformer architecture uses a conventional Encoder-Decoder structure. In particular, the architecture consists of a stack of N Encoders on the left-hand side of Figure 2.3, where input elements are uncorrelated, allowing parallelization. On the right hand side, a stack of N Decoders is shown, where inputs rely on the previous output prediction token. This creates a temporal dependence between inputs, and prohibiting simultaneous output of prediction results.

Transformer architecture simplifies the loop structure, which makes it impossible to directly capture position information in a sequence. To overcome this limitation, Transformer uses positional encoding to append positional information to each input vector, allowing the model to better grasp the sequential information in the sequence. The basic architecture is shown in Figure 2.3.

Transformer architecture suffers from a significant drawback where identical tokens within the input corpus are assigned identical embeddings due to the absence of a loop structure. To address this limitation and ensure that the relevant positional information of tokens in the sequence is captured, *Vaswani* et al. introduced positional encoding after the embedding stage.

Position encoding is a technique used to incorporate positional information into sequence input embeddings. This technique involves adding an offset vector to the input embedding vector. The offset vector is generated by a specific function that generates a unique position encoding at different locations and time steps. The most commonly used functions for generating position encoding are the sine and cosine functions. These functions are effective in capturing relative position information between different locations in a sequence and representing it as a continuous vector.

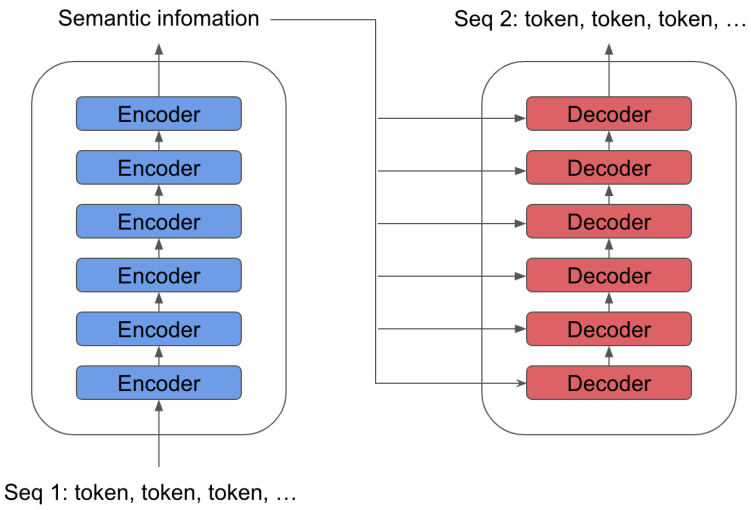

Figure 2.3: Transformer Architecture, on the left is stack of encoders, on the right is stack of decoders.

Positional encoding with sine and cosine is done using the following equations

$$\begin{aligned} PE_{(pos,2i)} &= \sin\left(\frac{pos}{10000^{2i/d}}\right) \\ PE_{(pos,2i+1)} &= \cos\left(\frac{pos}{10000^{2i/d}}\right). \end{aligned} \tag{2.4}$$

The Transformer uses the sine function to encode the token in even positions and the cosine function to encode the position in odd positions.

2.1.4 BERT

BERT used the Transformer encoder for stacking, with Google offering two variants: $BERT_{base}$, featuring a 12-layer encoder, and $BERT_{large}$, containing a 24-layer encoder. These models also exhibit different attention head quantities and different hidden layer parameters. $BERT_{base}$ has 12 attention heads and 768 hidden layers, while $BERT_{large}$ has 16 attention heads with 1024 hidden layers.

Figure 2.4 shows the architecture of the $BERT_{base}$ model used for sequence classification. It consists of three layers: Token embedding layer, Transformer encoder layer, and a task-specific output layer.

The Token embedding layer is responsible for creating embeddings that contain both sentence and position information for existing tokens. The Transformer encoder layer, which is built by stacking multiple encoders, extracts semantic information from the input and passes it to the task-specific output layer.

The task-specific output layer generates the final output based on the requirements of the downstream task. By selecting different outputs, various downstream tasks can be performed. For a sequence classification task, a dense layer is randomly initialized as classification header at the output of the encoder.

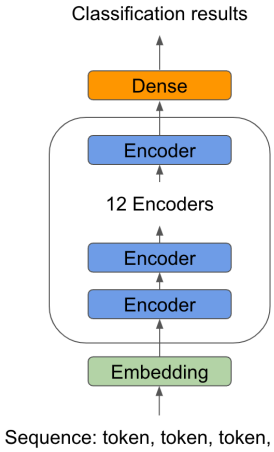

Figure 2.4: $BERT_{base}$ for sequence classification

2.1.5 Pre-train and Fine-tuning

The pre-training procedure for BERT involves two primary components. The first component is called Masked Language Modeling (MLM), which involves masking a segment of words in the training corpus and training the model to predict missing words based on the context of surrounding words. This approach enhances the model's ability to understand the semantics of masked words by leveraging the context of the words to the left and right of the masked word. This results in a bidirectional understanding of the language model, which allows it to comprehend language more effectively.

The second component of BERT pre-training procedure is known as the Next Sentence Prediction (NSP) task. The NSP task involves training the model to predict whether sentence B follows sentence A in a given text, with a 50% probability that B is the next sentence. By training the model on this task, BERT is better able to understand the relationship between sentences in a document, which is essential for many natural language processing tasks, such as question answering and sentiment analysis. The combination of MLM and NSP enables BERT to learn rich representations of language that can be fine-tuned for a wide range of downstream tasks.

The term "fine tuning" refers to the process of optimizing a pre-trained model on a specific dataset and usually downstream task. This technique differs from pre-training, which involves training a model on a large dataset to learn general features that can be applied to various tasks. During this phase, the model architecture undergoes rapid changes.

Once the model architecture is fixed, we can accelerate the training process by using a custom kernel. Therefore, determining the model architecture is critical to determining the optimization tools that can be applied.

The primary goal of fine-tuning is to retain the knowledge acquired by the pre-trained model while adapting it to the target dataset and downstream task. To

achieve this, several techniques are used to prevent the model from forgetting previously learned features. For example, when doing fine-tuning on computer vision task e.g. VGG network [21], it may beneficial to freeze its feature extraction layer and train only the dense layer for classification on a new dataset. This approach reduces computational costs, but it also has the disadvantage of limiting the model's ability to achieve high accuracy.

The learning rate is a critical parameter in the model fine-tuning process, and common techniques include lowering the learning rate, adjusting the optimizer's β, using warm-up, and learning rate decay [22]. These methods aim to strike a balance between model training stability and learning speed during the fine-tuning process.

In this thesis, the words "training" and "fine-tuning" are used interchangeably, if any section, paragraph, etc refers to pre-training then "pre-training" is explicitly used.

2.2 Deep Learning Optimizer

Deep learning optimizer is a set of algorithms used to adjust the weights and biases of a deep neural network throughout the training phase with the aim of reducing the error between the predicted and true values. Essentially, the main task of the optimizer is to determine the set of parameters that will produce the most accurate model predictions on the training data and to update the model parameters according to a selected algorithm.

The optimization process involves iteratively modifying the model parameters by using the gradient of the loss function associated with those parameters. The optimizer determines the direction and magnitude of each parameter update, while the learning rate controls the size of the update. Various optimization algorithms use different techniques to compute parameter updates, which may affect the speed and accuracy of model training.

2.2.1 Stochastic Gradient Descent

Stochastic gradient descent (SGD) [23] is a commonly used and straightforward method for optimizing model parameters in machine learning. Its popularity is mainly due to its intuitiveness and ease of implementation, making it a popular choice among researchers and engineers.

Denoting the model parameter set at iteration t by θ_t, the fundamental concept behind stochastic gradient descent is illustrated by the equation

$$\theta_{t+1} = \theta_t - \eta_t \nabla L(\theta_t)\,, \tag{2.5}$$

where η_t is the learning rate and $\nabla L(\theta_t)$ denotes the gradient of the loss function with respect to the parameter vector θ_t, evaluated for the i-th sample randomly chosen from the training set.

As machine learning progresses, researchers have discovered that the choice of learning rate can significantly affect the convergence and dispersion of the model. If the

learning rate is set too low, the model is guaranteed to converge, but the learning speed is impractically slow. Conversely, if the learning rate is set too high, the model can learn features from the training data more quickly, but it may struggle to converge to a local optimum.

To solve this problem, researchers proposed dynamic learning rate algorithm [24], which is an extension of the stochastic gradient descent algorithm. This approach adjusts the learning rate as a function of the number of iterations, resulting in a schedule of learning rates. During the first iteration, the algorithm induces a large change in the parameters of the model, while subsequent iterations incur smaller and smaller changes.

2.2.2 Momentum

Momentum is an optimization concept in machine learning that helps to accelerate the gradient decedent in the correct direction, thus speeding up the convergence rate. It is an extension of the gradient descent optimization algorithm that builds historical memory information in the search direction to overcome the local minima and oscillations of noisy gradients. Momentum is based on the same concept of momentum in physics, where a ball rolling down a hill builds enough momentum to jump out of the local minimum and bring it to a global minimum. The main idea behind momentum is to calculate an exponentially weighted average of historical gradients and use it to update the weights. By considering the past gradients, the gradient descent step becomes smooth, which reduces the amount of oscillations seen in the iterations.

$$\begin{aligned} g_t &\leftarrow \nabla L(\theta_{t-1}) \\ v_t &\leftarrow \gamma v_{t-1} + \eta_t g_t \\ \theta_t &\leftarrow \theta_{t-1} - v_t \end{aligned} \tag{2.6}$$

Equation 2.6 shows how the momentum is involved in the gradient update. The g_t represents the loss resulting from the calculation of the result from the current parameter versus the actual result, which is the gradient that the sample tells us to discard. The v_t represents the momentum accumulation, i.e. the gradient of the current term is weighted and added to the gradient of the previous steps so that historical gradient information can be introduced to help the system maintain the correct gradient direction. Where γ represents the weight of the momentum in the previous steps and η represents the learning rate of the current gradient. Finally, the parameters are updated using the v_t term to obtain the new parameters.

2.2.3 Adam

The Adam optimizer combines the ideas of both momentum and adaptive learning rate, which allows the Adam optimizer to accelerate the training process of deep learning models and avoid some problems of SGD algorithm, such as the learning rate is difficult to adjust and easily fall into local optimal solutions.

$$
\begin{aligned}
g_t &= \nabla L(\theta_t) \\
\theta_t &= \theta_{t-1} - \frac{\alpha \cdot g_t}{\sqrt{\sum_{i=1}^{t} g_t^2}}
\end{aligned}
\tag{2.7}
$$

The following discussion focuses on the concept of adaptive learning rate, which originated from the AdaGrad optimization algorithm. To address the limitations of the stochastic gradient descent (SGD) and momentum algorithms, AdaGrad was introduced by John Duchi et al. in 2011. AdaGrad enables the adjustment of different learning rates for each parameter and updates frequently changing parameters with smaller steps, while sparse parameters are updated in larger steps.

The adaptive learning rate in AdaGrad is determined by the sum of the squares of past gradients g_t up to the current time step t. If there are many parameters being updated in g_t (non-sparse), the sum of the squares will be larger, leading to a decrease in the learning rate for the current θ_t. Conversely, if there are fewer parameters being updated in g_t (sparse), the sum of the squares will be smaller, resulting in an acceleration of the update of the θ_t. This relationship is expressed in equation 2.7.

$$
\begin{aligned}
g_t &= \nabla L(\theta_t) \\
m_t &= \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t \\
v_t &= \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2 \\
\hat{m}_t &= m_t / (1 - \beta_1^t) \\
\hat{v}_t &= v_t / (1 - \beta_2^t) \\
\theta_t &= \theta_{t-1} - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t
\end{aligned}
\tag{2.8}
$$

The Adam algorithm workflow is presented in Equation 2.8. The algorithm incorporates momentum and adaptive learning rates to accelerate the machine learning model's training process and avoid local optima. The momentum term, m_t, controls the gradient descent's direction, preventing the model from falling into a local optimum. The v_t term adjusts the learning rate, η, based on the gradient's sparsity. By combining adaptive learning rates and momentum, Adam can optimize the model's parameters efficiently.

The weights of the first-order and second-order moment estimation are controlled by β_1 and β_2, respectively. β_1 is the exponential decay rate of the first-order moment estimation, and its value is usually set to 0.9 to balance the weight of the historical gradient. Similarly, β_2 is the exponential decay rate of the second-order moment estimation and is usually set to 0.999 to adjust the squared weight of the historical gradient.

2.2.4 Adan

Xie et al. [25] recently proposed Adan, an optimization algorithm that employs the Nesterov momentum [26]. By leveraging newer weights, Adan is able to anticipate the direction of gradient descent and determine the final gradient update direction, which can potentially accelerate convergence compared to using momentum alone. However, a potential limitation of Nesterov momentum is the need to incorporate the yet-to-be-updated momentum into the network for propagation to the loss value, resulting in multiple recomputations. As a consequence, the full potential of Nesterov momentum has not been realized, and further development of the method is required.

$$\begin{aligned} m_t &= \beta m_{t-1} + \nabla L(\theta_{t-1} - \alpha\beta m_{t-1}) \\ \theta_t &= \theta_{t-1} - \alpha m_t \end{aligned} \tag{2.9}$$

Equation 2.9 requires the computation of ∇L to be performed beforehand to obtain the Loss value. However, it is possible to modify the equation so that the stored historical gradient can replace the calculation process without altering the result.

$$\begin{aligned} m_t &= \beta m_{t-1} + \nabla L(\theta_{t-1}) + \beta[\nabla L(\theta_{t-1}) - \nabla L(\theta_{t-2})] \\ \theta_t &= \theta_{i-t} - \alpha m_t \end{aligned} \tag{2.10}$$

Equation 2.9 can be modified to obtain equation 2.10. This modification is equivalent and does not affect the convergence of the training. Furthermore, it becomes apparent that $\nabla L(\theta_{t-1}) - \nabla L(\theta_{t-2})$ approximates the second-order derivative of the objective function. By leveraging this information the direction of the gradient change can be determined, thus achieving faster convergence compared to stochastic gradient descent.

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2)(g_t - g_{t-1}) \\ n_t &= \beta_3 n_{t-1} + (1-\beta_3)[g_t + \beta_2(g_t - g_{t-1})]^2 \\ \eta_t &= \eta/(\sqrt{n_t} + \epsilon) \\ \theta_{t+1} &= (1+\lambda_t\eta)^{-1}[\theta_t - \eta_t \circ (m_t + \beta_2 v_t)] \end{aligned} \tag{2.11}$$

The basic structure of Adan is presented in equation 2.11. Notably, the authors have implemented first-order and second-order moment estimations which are similar to those employed in Adam's optimizer. To achieve Nesterov momentum, it is necessary to calculate with g_{t-1} subsequent to acquiring the gradient $g_{t-1} - g_{t-2}$.

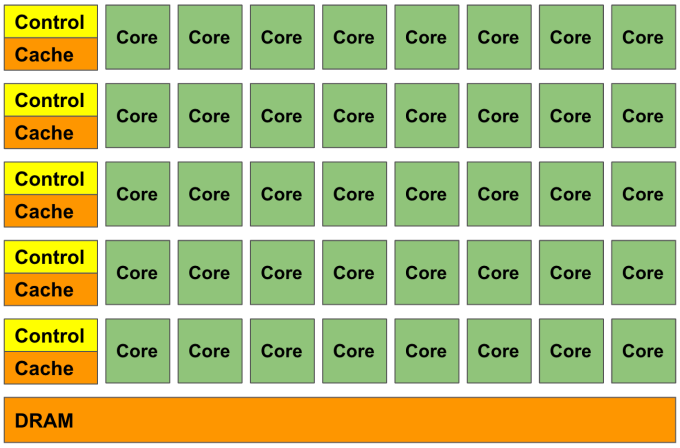

Figure 2.5: GPU architecture example

2.3 Hardware Architecture

This section is dedicated to explaining the design and ML application of existing hardware. How the hardware works, and what benefits it provides when training deep neural networks.

2.3.1 GPU

The following section concentrates on the use of graphics processing units (GPU) and will not delve into traditional graphics processing units. Due to the advancement of a sophisticated programming model, general-purpose computing can now be implemented with GPUs. Contemporary GPUs feature a "manycore" design, known as Streaming Multiprocessors (SM) by NVIDIA and Compute Units (CU) by AMD. These SM cores execute a single instruction multiple thread (SIMT) program initiated by a corresponding kernel. Additionally, multiple threads can be run on each SM, allowing for communication via scratchpad memory and synchronization through barrier operations. Each core typically contains a first-level instruction and data cache, which decreases the amount of data sent to lower memory levels. The vast number of threads available can be utilized to hide the latency caused by cache misses.

Figure 2.5 shows a group of Streaming Multiprocessors or Compute Units that share the same cache and control unit. Such a group of compute units is hierarchical in the GPU programming model, where NVIDIA refers to it as warp and AMD refers to it as wavefront. SMs/CUs execute the same instruction using different data, which significantly enhances the data throughput of the instruction. In contrast, a CPU may not match this throughput gain, even with vector instructions.

However, GPU's throughput decreases substantially when a set of SMs/CUs produces different branch in conditional statements. To handle branches, modern GPUs insert instructions back into the instruction stream and fetch them multiple times until all result were calculated, using a mask to prevent some threads from writing

results to registers to complete the branching operation. As a result, an excessive number of branches in an instruction degrades the performance of the GPU. In contrast, a CPU has a more robust control unit that enhances its ability to handle instruction branches.

The high degree of parallelism and the hierarchical memory architecture of GPU present possibilities for optimization of existing machine learning software. The pipeline structure of GPU compute core allows for commonly used pipeline optimization techniques such as loop unrolling and instruction rescheduling to improve pipeline performance. Additionally, special vector instructions can be employed to enhance data reading capabilities. The hierarchical cache structure enables caching of frequently used data, reducing unnecessary memory access and improving performance.

Despite advancements in computer systems, the CPU remains at the center and is referred to as the host, while GPUs and other accelerators are regarded as devices, functioning as subordinate peripheral devices. To use a GPU, engineers must manually specify programs to run on it, which results in additional overhead. Therefore, reducing this overhead is a crucial topic for optimizing modern GPU programs.

2.3.2 TPU

Tensor Processing Unit (TPU) [27] is a specialized accelerator for Tensor operations, which was introduced by Google in 2017. Its primary function is to facilitate the efficient execution of matrix operations that are pervasive in deep learning. The TPU is a custom designed chip that consists of several dedicated components, such as a matrix multiplication unit, an accumulator, a nonlinear activation unit, and on-chip buffer.

The Matrix-multiply units (MXU) is the foundation of the TPU, which presents a different architecture compared to a GPU. While a GPU is often described as temporal architecture, TPU is characterized as spatial architecture, both of which employ similar computing units and multiple units as computing components. TPU's spatial design enables it to support a variety of datapaths, facilitating data exchange across multiple Processing Elements (PEs).

The Systolic array is a classical architecture in spatial architecture that was first introduced by *Kung et al.* [28] . This architecture is characterized by its simplicity and low cost. The primary design goal of the systolic array is to reduce the overhead of data transfer between memories by extending the data flow time in the computing unit. The systolic array achieves this by using data as many times as possible in the process pipeline, fetching data once, and cascading multiple PEs to use the results of the previous level of operations as input to the current PE.

As shown in Figure 2.6, the first data enters the first PE and is processed, then passed to the next PE. Meanwhile, the second data enters the first PE, and so on. By the time the first data reaches the last PE, it has been processed multiple times. This pulsating architecture actually reuses input data multiple times, resulting in high computing throughput while consuming less memory bandwidth. Upon expan-

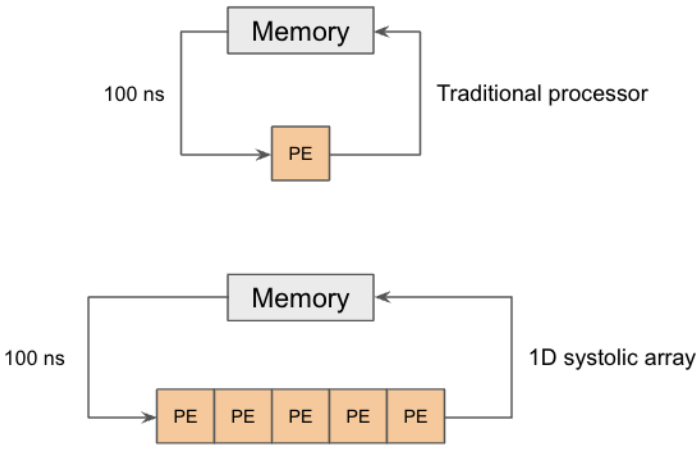

Figure 2.6: Principle of systolic array

sion of the one-dimensional array shown in Figure 2.6 into a two-dimensional array, the resulting structure is recognized as the matrix multiplication unit (Figure 2.7) present in TPU.

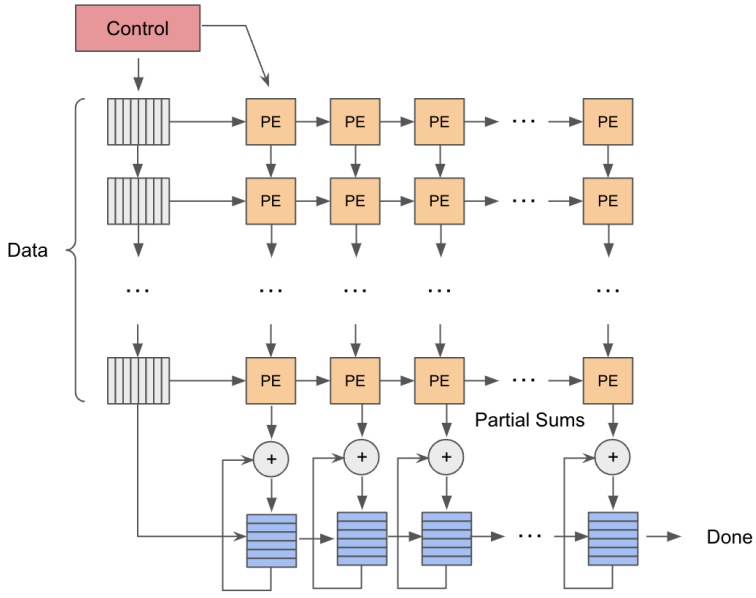

Figure 2.7: MXU architecture

In contrast to GPUs, TPUs utilize a systolic array, which is a data-flow dependent computational architecture. This unique structure takes advantage of potential parallelism in computationally intensive tasks, allowing for high throughput even at lower clock frequencies. By calculating the dynamic energy consumption using the formula $E = c \cdot f \cdot V^2$ it's possible to see that operating at a lower circuit frequency at a certain voltage leads to lower power consumption. Google's research [29] shows that a computing cluster of 16 TPUv2 chips can consume 36% less energy

compared to a cluster of 16 V100 32GB GPUs.

The TPUv2 has a power consumption of 280W per chip and contains two MXU cores on each chip. The instance we have access to contains four TPUv2 chips, resulting in an available system power consumption of about 1120W. In comparison, the average power consumption of the NVIDIA V100 PCIe 16GB is about 250W per chip.

In summary, the goal is to compare TPUs with GPUs in terms of their training time, and cost of use. Furthermore, to assess the level of complexity involved in transitioning from GPU training to TPU training, including aspects such as software reliability and usability. This comparison will give users more insight into choosing the proper configuration to accelerate their machine learning tasks.

2.4 Mixed Precision Training

Mixed precision training [12] is a popular technique that exploits the low precision of FP16 to reduce computation and memory overhead while maintaining model accuracy. The key idea is to store network parameters in the low-precision FP16 format while maintaining accuracy in the high-precision FP32 format. During training, the range of parameter values is usually kept within a limited range, which makes the use of FP16 parameters a viable option with little impact on accuracy. However, the gradient has the potential to suffer from numerical underflow, especially in the later stages of training, resulting in a loss of accuracy. The basic weight update process is illustrated in the Figure 2.8.

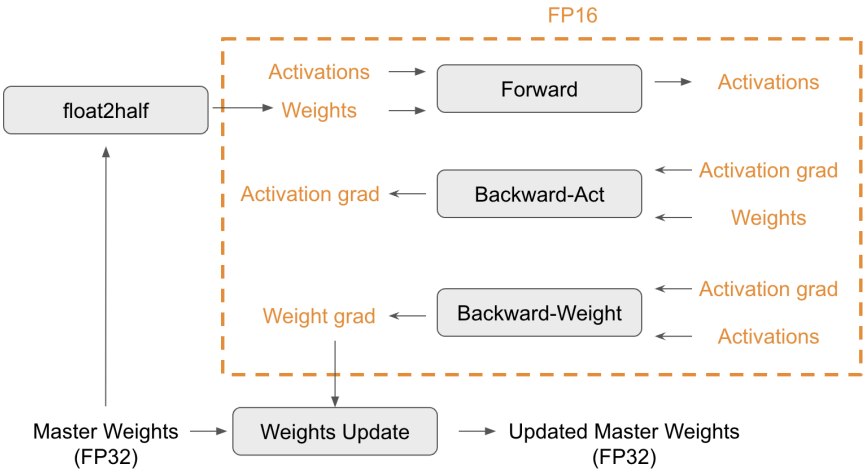

Figure 2.8: Mixed precision training weight update

To alleviate this problem, a technique called gradient scaling is introduced, where the FP16 gradient is multiplied by a loss scale mapped to a range of values suitable for FP16 operations to avoid overflow before being used for backpropagation. At the end of the backpropagation the gradient is divided by the loss scale so that the true value of the gradient can be restored.

Figure 2.9: BF16, FP16 and FP32 formats. Different bit ranges for above three floating number representations.

If the gradient is found to have underflowed during the update process, then the update is skipped. In the parameter update step, all updates are performed on the FP32 copies of the model parameters to ensure the stability of the values. This approach results in faster training time and less memory consumption due to the use of less accurate FP16 parameters, while the use of more accurate FP32 parameters ensures accurate parameter updates.

FP16 and BF16

Mixed-precision training gains a significant advantage from low-precision floating-point representations, such as FP16 and BF16. FP32 and FP16 are commonly used in GPU training, while BF16 is preferred for Google's TPU training. Both FP16 and BF16 use 16-bit data, with differences in the length of their mantissa and exponent, as illustrated in Figure 2.9. Note that BF16 has the same exponent bits as FP32, so they have the same representation range as FP32.

Converting from generic FP32 data to BF16 is simpler, as the exponent of BF16 is the same as that of FP32, requiring no special processing. However, converting from FP32 to FP16 requires a special process due to the difference in their precision representation capabilities. Typically, FP32 needs to be rescaled to adapt to the precision of FP16.

2.5 LightSeq Library

LightSeq, originally created for high performance Inference using Transformers has been extended, under the name of LightSeq to accelerate the training / fine-tuning process of Transformer-based models on GPUs [30], [31]. As previously stated, Transformer-based models use is widespread due to their stellar performance in many ML tasks with one of the most common approaches being fine-tuning the likes of large pre-trained models such as BERT, GPT, etc.

LightSeq comes with a series of optimization techniques for GPUs specifically tailored for Transformer-based models computation and memory access flows in order to improve performance. LightSeq focuses on following optimization techniques: Computational Graph Optimization, Dependent Reduction Rewriting, Accelerated Mixed-Precision Update for Trainer and Dangling-Tensor Aware Memory Manager.

Computational Graph Optimization - LightSeq aims to replace as many fine-grained GPU kernel operations implemented in PyTorch with coarse-grained fused ones, which reduces overhead caused by launching kernel functions. General matrix multiply (GEMM) operations such as linear transformation and scaled dot product use the implementations found in cuBLAS, whilst non-GEMM operations such as Dropout, LayerNorm and Softmax are re-implemented in LightSeq. Element-wise operations such as DropOut, ReLU, etc are fused into coarse-grained kernels if theyre adjacent in the computational graph of the Transformer. These operations also enable more optimizations in the shape of parallelism. Layers such as the embedding layer and criterion layer are also optimized with more parallelism, and Softmax is also reworked such that instead of being directly calculated, the logarithmic Softmax is calculated instead.

Accelerated Mixed-Precision Update for Trainer - Parameters and gradients are FP16 during forward- and back-propagation when using mixed precision, however the optimizer requires FP32 copies to ensure accuracy. Traditionally each gradient or parameter is copied and stored as a FP32 copy, and then loaded when needed by the optimizer. This approach introduces two inefficiencies; mass-kernel launches when copying and updating which reduces GPU utilization as well as redundant memory footprints (storing FP16 and FP32 gradients/parameters) which reduces memory efficiency. LightSeq resolves these two inefficiencies by using symbolic tensor linking and on-the-fly conversion. When initializing the optimizer, parameters and gradients are copied into a separate workspace tensor, which is then linked to the optimizer. LightSeq only executes the optimizer update one time per training step to update the workspace, instead of for every updated parameter/gradient which reduces kernel launches. When loading from the workspace, the parameters/gradients are loaded as FP16, then converted on-the-fly to FP32 in the registers to be updated. Once updated, theyre re-converted to FP16 before being stored in the workspace. This reduces the data movement by 50% and avoids redundant FP32 copies since all memory access is done using FP16.

Dependent Reduction Rewriting - Batch reduction operations such as LayerNorm and Softmax traditionally take a lot of time during training and are therefore a focus of optimization in LightSeq. Changes to how the batch reduction operations of LayerNorm as well as re-arrangement of gradient calculation formula allows for parallel execution of the batch reduction operations. LightSeq stores parameters as FP16 floating points but converts them to FP32 during computation due to sensitivity to low precision of floating point numbers when using LayerNorm.

Dangling-Tensor Aware Memory Manager - In general, large batch training leads to faster convergence and higher accuracy however due to memory limitations it requires either multiple GPUs, memory offload or gradient accumulation.

LightSeq divides GPU memory into permanent and temporary, to store parameters and gradients as well as intermediate tensors respectively. During the initialization phase LightSeq scans the training set and estimates the capacity required, then the temporary memory is allocated once and reused by different batches before finally released at the end of training. Which reduces the memory footprint by compacting the memory and reducing the allocation and releases.

2.6 Distributed Training

Distributed training is a technique used to address the problem of handling large volumes of data needed to train machine learning models, especially neural networks. Given the limited computing power of individual machines, it has become necessary to distribute machine learning workloads across multiple machines, transforming a centralized system into a distributed one. Two primary parallel approaches, data parallelism and model parallelism, are used to facilitate distributed training. These methods can be used independently or in combination. In addition, when working with distributed training, it is essential to consider the fault tolerance and scalability of the system.

2.6.1 Topology and Communication

Parameter Server [32] and Ring All Reduce [33] algorithms are widely used in industry, such as at Google or Microsoft. Parameter Server involves a master server and multiple worker nodes, and is therefore an asymmetric system where worker nodes obtain all parameters of the model from the master node through inter-node communication. Ring All Reduce utilizes a symmetric point-to-point system where all nodes are equal and form a ring structure. The transmission of node parameters in this system involves each node transmitting its parameters to the left node and receiving the remaining parameters from the right node. This process effectively reduces network communication, allowing all nodes to obtain updated values of model parameters during the transmission process.

There are two modes of inter-node communication when obtaining model parameters: synchronous and asynchronous. In a symmetric system, where each node has comparable computational speed, synchronous communication is preferred to maintain the model's accuracy. However, in systems with nodes of varying performance, synchronous communication can limit system throughput. In such scenarios, asynchronous communication can improve system throughput by reducing the waiting time of faster nodes. However, asynchronous communication introduces random noise (staleness), which represents an out-of-date direction of gradient descent. This noise can be equivalent to the momentum used in the optimizer [34], facilitating the parameter update process and preventing the model from falling into a local optimum.

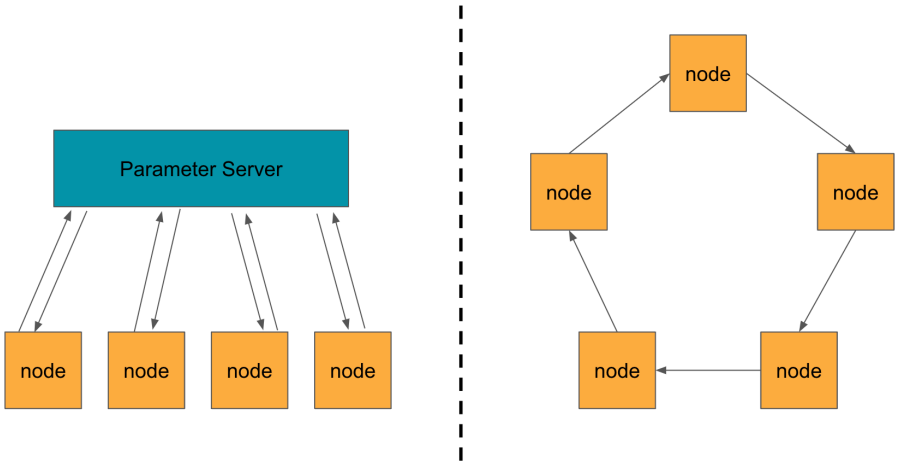

Figure 2.10: Distributed training topology, on the left is parameter server, on the right is ring organization.

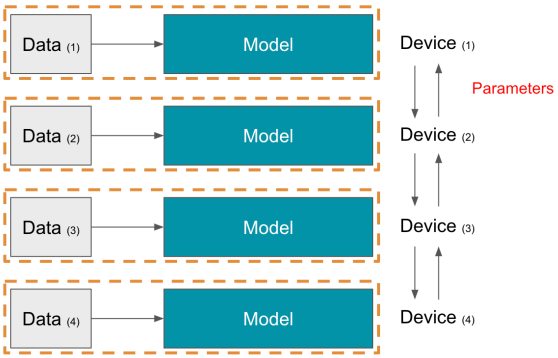

Figure 2.11: Data Parallel

2.6.2 Data Parallel

The data parallel approach involves replicating model parameters on each node in a distributed system. To facilitate training, the training data is divided into N copies, where N is equal to the number of nodes in the system. During the training process, each node updates the gradient information based on the current batch of data. An update mechanism, such as the Parameter Server or Ring All Reduce, is used to ensure that these updates are propagated throughout the system and applied to all copies of the model parameters.

As shown in the Figure 2.11, the training data is divided into four parts, and the entire model is stored on each device, using a quarter of the dataset for training, and updated parameters are transferred between the devices to ensure the gradient descent of the model parameters in the correct direction.

The use of data parallelism is a popular technique in distributed training, especially when the model size is within the limits of a single GPU's capacity. However, when the model size is too large to fit into a single GPU, alternative methods such as model parallelism or pipeline parallelism should be considered. These alternatives

come with added communication and scheduling overhead.

2.6.3 Workflow

The data parallel execution model currently in use operates as follows: The distributed initiator broadcasts the training data from the data sitting node to the entire communication network via Broadcast. Subsequently, each GPU starts the forward, which does not require communication between GPUs because there is no data dependency.

However, during the backward, gradient information from each layer needs to be exchanged with other devices so that the entire communication network obtains gradient accumulation information within the entire mini-batch. This step is known as reduce sum within the collective communication. In practice, communication is initiated as soon as a layer in the network completes backpropagation. This provides the advantage of overlapping communication with computation and completely masks the communication overhead.

In this parallel process of computation and communication, the slower communication process often determines the time overhead. This may result in completion of the calculation but waiting for the communication channel to become available, resulting in a cumulative delay in the backpropagation process. Once all devices have obtained the complete gradient information, averaging the gradient within the batch allows the optimizer to obtain the required gradient, and executing optimizer.step() completes the update process. Therefore, the bottleneck that affects the training speed throughout the training step is the communication time between GPUs and the computation time of backpropagation process.

2.6.4 Collective Communication

Collective Communications refers to a global communication process involving all processes in a particular process group. It consists of fundamental operations such as send, receive, copy, synchronize Barrier, and the synchronization of processes between nodes (signal+wait). These fundamental operations are combined to form communication templates, also known as communication primitives. The challenge of collective communication lies in ensuring communication efficiency and optimal application of the network hardware connection topology.

In the forward propagation of distributed training broadcast is mainly used to distribute the training data to each node. Figure 2.12 shows the process of Broadcast where the data owning node will communicate with every other node, once and only once. Thus the communication overhead of Broadcast is N, the number of nodes in the N communication world.

Allreduce is a collective communication operation that aggregates data from multiple processing units and combines this data into a global result, utilizing a designated operator. The resulting data is then distributed to all processing units. Allreduce performs in several stages. First, each participant disperses its vectors. Second, each

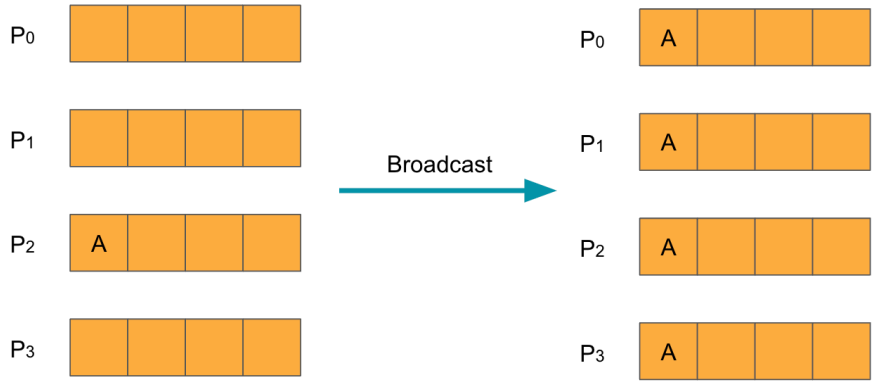

Figure 2.12: Broadcast communication

participant gathers vectors from other participants. Finally, each participant performs a preference operation on all accumulated vectors. By using various Allreduce sequences for different participants, highly complex computations can be distributed across many computational units.

Allreduce is widely used by parallel applications in high-performance computing (HPC) that relate to scientific simulation and data analysis, such as machine learning computations, as well as the training phase of deep learning neural networks. Given the substantial growth of deep learning models and the complexity of scientific simulation tasks employing networks, effective implementation of Allreduce is critical to reducing communication time.

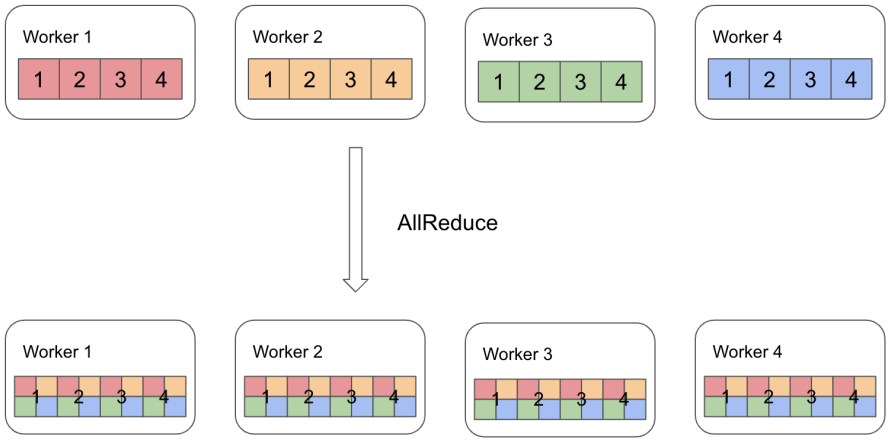

Figure 2.13: Allreduce communication

Assuming the existence of P processes and a total amount of N data. When the main process handles the data, the other processes send the data to the main process, resulting in $(P-1)N$ data received. After the main process finishes handling the data, it broadcasts the result to the other processes, resulting in $(P-1)N$ data sent. Therefore, the total amount of communication is $2(P-1)N$. This means that communication overhead will increase linearly with the increase in the number of P

processes.

In the Ring-AllReduce algorithm, the amount of communication per process can be calculated as follows. Each process must receive $\frac{N}{P}(P-1)$ data from the process on its left and send the same amount of data to the process on its right. Thus, the total communication is $2\frac{N}{P}(P-1)$. Chunking data into $\frac{N}{P}$ sizes distributes chunks to other processes, reducing communication bottlenecks. As the number of P processes increases, the total communication converges to $2N$, indicating that communication does not increase as the number of nodes in communication increases.

Therefore, the Ring-Allreduce algorithm is more efficient than the simple algorithm because it distributes computation and communication evenly among all participating processes, eliminating bottlenecks. Ring-AllReduce is commonly used in many AllReduce implementations, especially for distributed deep learning workloads.

3

Methods

This chapter explores various strategies for optimizing performance when fine-tuning BERT models. Specifically, we examined the Adan optimizer and implemented a hardware-optimized version using Kernel Fusion, Multi-tensor access and Instruction Level Parallelism (ILP). In addition, this chapter investigates the effects on fine-tuning performance when using techniques such as mixed precision training, distributed training, and LightSeq's acceleration library. Finally, all optimization techniques explored are used in a combined search with various hardware (NVIDIA A100, V100, A10, T4, and Google TPUv2, TPUv3) to find optimal solutions.

3.1 Fused Adan

Adan optimizer is a novel optimizer that diverges from the widely used Adam optimizer. It leverages the Nesterov momentum to accelerate the gradient descent algorithm, resulting in faster model training. As the algorithm is still in its early release, there is currently no optimized kernel available on the GPU. The implementation of fused Adan will be introduced in the following subsection.

3.1.1 Profiling

The current Adan kernel is not equipped with a high-performance fused kernel, and its original implementation is based on the tensor compute interface provided by Pytorch. Unfortunately, this implementation may result in potential performance losses. Therefore, the original Adan implementation is profiled to investigate potential issues, see Figure 3.1

By measuring the time consumed by optimizer.step() during one optimization step it is possible to examine how much time is consumed for computation tasks and data preparation tasks. The optimizer single step profiling resulted in a total time of 232.866 ms. This result can be broken down into the two parts: computation and data preparation, which take 169.331 ms and 63.535 ms, respectively.

The profiling reveals that there are 4422 kernel launches during one optimization step, with 804 kernels dedicated to data preparation and 3618 kernels used for computation. Additionally, a kernel launch time graph for this original implementation is provided in Figure 3.1. This figure illustrates that the kernel is arranged in a loose

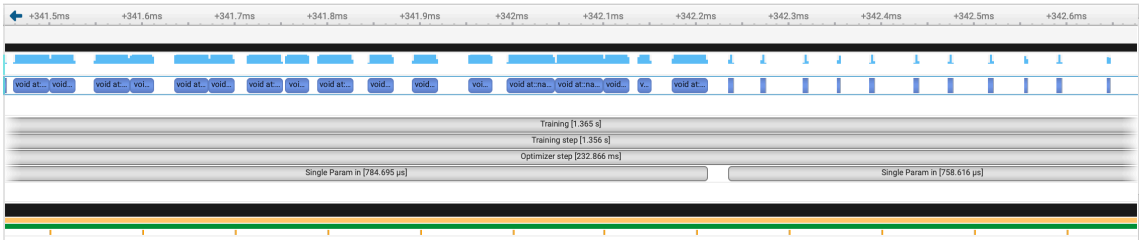

Figure 3.1: Unfused Adan kernel launch time graph with single tensor access.

manner on the timeline which results in a significant amount of GPU time being wasted, showing potential opportunities for optimization.

According to the profiling results the Adan optimizer takes a total computation time of 169331 μs. However, only $31,293.63$ μs is actually used for kernels running actively, which implies that the computing units of the device are active only during this time. As a result, only 18.48% of the total time is spent on actual computation. This suggests that there is a considerable amount of overhead that could be eliminated through optimization.

3.1.2 Kernel fusion

Kernel fusion [35] is a method used to eliminate the extra overhead caused by multiple kernel launches and might improve power efficiency on GPUs. By reusing the data which already loaded into register or shared memory, redundant data load/store could be reduced. A kernel is essentially a function call, such as torch.mul(), torch.add(). To illustrate, consider the example code below. For example, calculating $\mathbf{R} = \mathbf{A} \times \mathbf{B} + \mathbf{C} \times \mathbf{D}$ using the PyTorch implementation requires dividing it into three tensor operation kernel launches.

```
# Unfused version

# Create Variable, Matrix size (512, 512)
A = torch.rand(512, 512).to("cuda")
B = torch.rand(512, 512).to("cuda")
C = torch.rand(512, 512).to("cuda")
D = torch.rand(512, 512).to("cuda")

# Calculate R=A*B+C*D, with intermediate variables M, N
M = torch.mul(A, B)
N = torch.mul(C, D)
R = torch.add(M, N)

# Calculate R=A*B+C*D, Fused version
fused_kernel(A, B, C, D, R)

/*Following code is written in CUDA style*/
__global__ void fused_kernel(float *A, float *B,
```

```
    float *C, float *D, float *R) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    if (row < 512 && col < 512) {
        float sum = 0;
        for (int i = 0; i < 512; i++) {
            sum += A[row * 512 + i] * B[i * 512 + col];
            sum += C[row * 512 + i] * D[i * 512 + col];
        }
        R[row * 512 + col] = sum;
    }
}
```

In the code above torch.mul() is used twice and torch.add() once, with each function call incurring a GPU kernel launch, leading to increased overhead when allocating resources, instruction scheduling and data transfer.

The fused Kernel combines two torch.mul() calls and one torch.add() call into a customized kernel, so that there only needs to be one kernel launch to perform the same operations as before.

3.1.3 Memory Hierarchy

In modern GPUs, multiple levels of storage are available for general-purpose computing, which typically includes four levels: DRAM, scratchpad memory (shared memory), local memory, and register. To access data, a thread must first locate it in DRAM and then copy it to its own storage, such as register or local memory. Local memory is physically located in DRAM, but since addresses can be determined at compile time, they can be cached. Register is preferred for frequently accessed data because it has the fastest access speed and is closest to the compute unit. However, register is a limited resource due to hardware constraints. Scratchpad memory can be used as an alternative and has similar access speeds to register.

The fused kernel Adan implementation avoids excessive access to device DRAM by using registers to store frequently access variables, which can lead to memory walls. This technique can significantly improve the arithmetic intensity at DRAM level and reduce GPU idle time.

3.1.4 Vector Instruction

Vector instructions are a type of instruction that operates on multiple data elements in a single operation, derived from the Single Instruction Multiple Data (SIMD) model. In addition to their use in multimedia processing, such as image and audio/video processing, scientific simulation, and machine learning, vector instructions are widely used in high-performance computing due to their ability to reduce the number of instructions needed to operate on large data sets, resulting in increased throughput.

NVIDIA supports vector instructions in its Parallel Thread Execution (PTX) instructions, such as vectorized Load and Store instructions. Using vector loading instructions reduces the total number of instructions, reduces latency, and improves bandwidth utilization. However, the corresponding instruction cycles are longer than non-vectorized instructions, making this trade-off beneficial for medium-performance workloads.

```
/*0088*/                   IMAD R10.CC, R3, R5, c[0x0][0x140]
/*0090*/                   IMAD.HI.X R11, R3, R5, c[0x0][0x144]
/*0098*/                   IMAD R8.CC, R3, R5, c[0x0][0x148]
/*00a0*/                   LD.E.64 R6, [R10]
/*00a8*/                   IMAD.HI.X R9, R3, R5, c[0x0][0x14c]
/*00c8*/                   ST.E.64 [R8], R6
```

As an example, the code above utilizes the IMAD instruction to compute the address of the vector load. It subsequently uses LD.E.64 to load two 32-bit floating-point numbers into the R6 register simultaneously. The IMAD instruction is then employed again to calculate the data storage address, and the ST.E.64 instruction is used to store the data into the address indicated by the R8 register simultaneously.

Vector loading can be utilized by using vector data types defined in the CUDA C/C++ standard header, such as int2, int4, or float4. When using the fused Adan single tensor access kernel, the float4 type is used for processing FP32 data. As a result, the compiler is enabled to execute vectorized load instructions, such as LD.E.128, to retrieve the target data in one go instead of employing four LD.E instructions for data access. The following code provides further insight into the fused Adan kernel.

```
float4* p4_ptr = reinterpret_cast<float4*>(p);
/*Variable pointer...*/
float4* exp_avg_diff4_ptr = reinterpret_cast<float4*>(exp_avg_diff);

/*Vectorized load from global memory*/
float4 p4 = p4_ptr[global_id];
float4 g4 = g4_ptr[global_id];
const float4 neg_grad4 = neg_grad4_ptr[global_id];
float4 exp_avg4 = exp_avg4_ptr[global_id];
float4 exp_avg_sq4 = exp_avg_sq4_ptr[global_id];
float4 exp_avg_diff4 = exp_avg_diff4_ptr[global_id];
```

3.1.5 Instruction Level Parallelism

The concept of instruction-level parallelism plays a critical role in computer architecture, enabling a higher number of instructions to be executed per cycle by leveraging microprocessor pipeline architecture and minimizing pipeline bubbles through instruction scheduling. The fused adan multi tensor access kernel extends instruction scheduling space by utilizing loop unrolling techniques, allowing the compiler to

reorder instructions based on pipeline characteristics during the compilation stage for optimal hardware performance.

Loop unrolling has several benefits, including reducing the overhead caused by branch prediction failures and minimizing the number of variable maintenance instructions required for loop counters. For example, if the loop $for(i = 0; i < 100;)$, executing $i+ = 2$ instead of $i+ = 1$ reduces the number of ADD instructions required to maintain the variable i value from 100 to 50.

In addition, unrolling the loop allows processing of two iterations at a time, optimizing instruction space to make full use of the pipeline to cover access latency and improve throughput, especially when dealing with time-consuming access tasks such as data loading and storage within the loop body.

The provided code demonstrates the implementation of loop unrolling in fused Adan multi tensor access. Specifically, by setting the ILP to 4 for each thread, the loop is processed with four elements at a time. The compiler is instructed to perform loop unrolling using the *#pragma unroll* directive, which enhances both code readability and efficiency.

```
#define ILP 4
#pragma unroll
for(int ii = 0; ii < ILP; ii++)
    {
        int i = i_start + threadIdx.x + ii*blockDim.x;
        if(i < n && i < chunk_size)
        {
          r_p[ii] = p[i]; // r_ means register variable
          r_g[ii] = g[i];
          r_exp_avg[ii] = exp_avg[i];
          r_exp_avg_sq[ii] = exp_avg_sq[i];
          r_exp_avg_diff[ii] = exp_avg_diff[i];
          r_neg_grad_diff[ii] = neg_grad[i];
        } else {
          //...
        }
    }
```

3.2 Multi Tensor Access

Multi Tensor Access is a grouping technique that improves the device's memory access efficiency and results in significant performance improvements. In situations where a multitude of small tensors exist, it is possible to have the process by aggregating the individual tensor accesses into a larger one, see Figure 3.2. This greatly simplifies the operation, leading to reduced kernel launches and global memory access, ultimately improving performance.

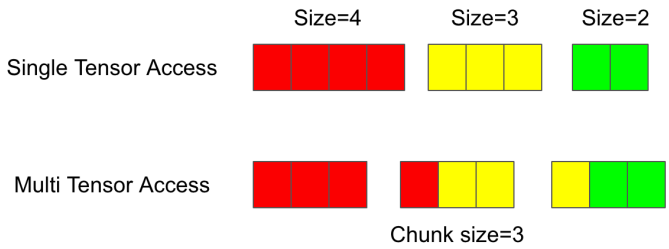

Figure 3.2: Multi Tensor Access, with a chunk size of 3. Meaning that the tensors (red, yellow and green) are split into chunks of 3 for more efficient memory accesses.

3.2.1 Profiling

Table 3.1 provides the SM and memory throughput for different kernel sizes when using the single tensor access fused Adan kernel.

Grid Size	Block Size	SM Throughput	Memory Throughput
21747	1024	39.45%	82.44%
2304	1024	37.34%	79.22%
576	1024	31.44%	62.96%
2	1024	0.78%	0.89%
1	1024	0.45%	0.50%

Table 3.1: Typical kernel size with their SM/Memory throughput

Figure 3.3 presents the SM and Memory throughput rate curves for running the Adan kernel. The results show that the use of single tensor access kernel results frequent kernel launches with smaller sizes, leading to a significant reduction in both SM and memory throughput.

By observing the data in Table 3.1, a sharp decline in both computational and memory throughput is noticed as the grid size decreases. Therefore, it is beneficial to implement multi-tensor access to mitigate any potential variations caused by the kernel size. This will not only enhances the efficiency of the system but also ensures consistent performance.

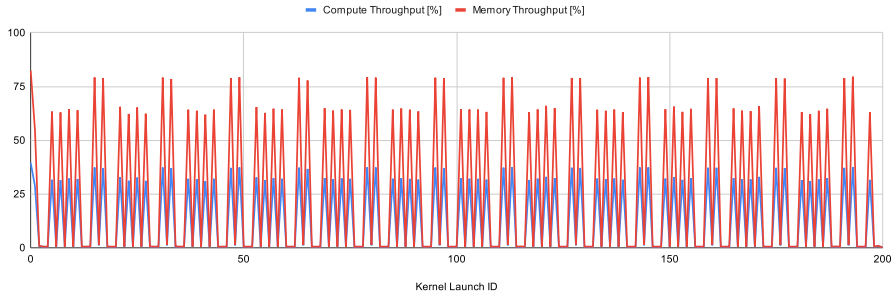

Figure 3.3: 1-Step benchmark displaying the GPU's SM and memory throughput when using Single Tensor Access.

3.2.2 Multi Tensor Apply

In order to use the Pytorch Optimizer interface, it is necessary to specify the foreach parameter that determines the mode in which parameters are accessed for optimization. When using a False foreach value, the optimization process iterates through each parameter tensor individually, leading to the sequential updating of these individual tensors. However, when using a True foreach value, the optimizer groups the parameter tensors according to a defined chunk size before sending them to the kernel as a tensor list.

To enable multi tensor access the tensors need to be packed before launching the kernel computations. To reduce startup overhead caused by many small tensors, multiple tensors are bundled into fixed-size chunks for computation, as depicted in Figure 3.2.

```
for (int t = 0; t < ntensors; t++) {
    tl.sizes[loc_tensor_info] = tensor_lists[0][t].numel();
    for (int d = 0; d < depth; d++)
        ...
        for (int chunk = 0; chunk < chunks_this_tensor; chunk++) {
        ...
            if (tensors_full || blocks_full || last_chunk) {
                multi_tensor_apply_kernel<<<loc_block_info,
                block_size, 0, stream>>>(
                    chunk_size, noop_flag.DATA_PTR<int>(), tl,
                    callable, args...);
            }
        }
    }
```

The first condition, **tensors_full**, requires the chunk sent to the kernel to be optimally sized for kernel operation. The second condition, **blocks_full**, ensures that the kernel operates optimally with the maximum allocated number of blocks. This prevents performance bottlenecks caused by too many or too few blocks. The last condition, **last_chunk**, handles tensors that cannot be processed under the first

two conditions, ensuring that all tensors are calculated without missing.

3.3 Mixed Precision Training

The current workflow does not utilize mixed-accuracy training, resulting in a missed opportunity for potential performance and energy benefits. The most time consuming GPU kernels during training are shown in Figure 3.4.

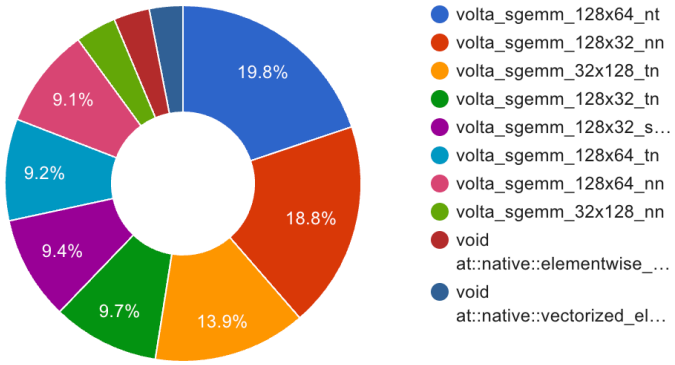

Figure 3.4: Profiling showing the most time consuming GPU kernels, during one training step when using the Hugging Face model without mixed precision.

In the realm of deep learning, the computation of matrix multiplication is acknowledged as a computation intensive kernel. Therefore, each of the Single precision General Matrix Multiply (sgemm) kernels illustrated in Figure 3.4 can be perceived as operators of high computational intensity. The sgemm kernel accounts for 93.6% of the overall time consumption in one training step. It follows that any acceleration in the execution of the sgemm kernel can lead to noteworthy enhancements in the overall training performance. Table 3.2 shows the execution time for some typical sgemm kernels with different matrix size.

Kernel	Mean Duration/μs
volta_sgemm_128x64_nt	1274
volta_sgemm_128x32_nn	1205
volta_sgemm_32x128_tn	743
volta_sgemm_128x32_tn	3113
volta_sgemm_128x32_sliced1x4_nt	3033

Table 3.2: Average execution time of sgemm kernels with different size when not using mixed precision.

3.3.1 Enable Mixed Precision Training

By introducing automatic mixed precision training (AMP) into the training flow, the process can be optimized in terms of time and energy consumption for better

results. This thesis will leverage Hugging Face's acceleration library to automate scaling of digital ranges and FP32-FP16 conversion. Minor modifications to the training loop are all that's required to enable automatic mixed precision training.

```
from accelerate import Accelerator
accelerator = Accelerator()
model, optimizer, train_loader, scheduler = accelerator.prepare(
            model, optimizer, train_loader, scheduler)
...
# Code in trainer
for step, batch in enumerate(self.train_dataloader):
    batch = {k: v.to(self.device) for k, v in batch.items()}
    self.model.zero_grad()
    outputs = self.model(**batch)
    loss = outputs.loss
    self.accelerator.backward(loss) # original: loss.backward()
    # ...
```

The code above demonstrates the initialization process required before using the accelerator. In this code, the loss parameter is passed to the backward function provided by the accelerator. To ensure proper functioning, it is crucial to initialize the model beforehand. During the initialization process, the accelerator object establishes internal links to the original model parameters. Once the loss is passed in, the FP16 model optimized by the accelerator can be used after range scaling.

Mixed precision training offers two main advantages, which are briefly described below,

1. **Reduced memory usage:** FP16/BF16 is half the length of FP32, meaning that more data can be transferred during training. Which may enable more data to be used per batch during training. However, depending on the implementation, mixed precision might yield higher requirements. E.g. Hugging Face's implementation of mixed precision stores FP32 copies of model parameters as well as stores gradients in FP32, whilst also storing the FP16 values for the model parameters. Meaning that more memory is actually required for storing model parameters and gradients. In particular, Hugging Face's implementation requires 4 bytes per model parameter as well as 4 bytes per gradient during FP32 training. Whilst requiring 6 bytes per model parameter and 4 bytes per gradient during mixed precision.

   This is unlike the LightSeq implementation, which avoids storing FP32 copies, and instead uses on-the-fly FP16 <-> FP32 conversion. Meaning that LightSeq reduces memory requirements by only storing 2 bytes per model parameters and even 2 bytes per gradient. Which yields, both a speedup since every access is FP16 as well as lower memory requirements.

2. **More computationally efficient:** Special AI acceleration chips such as the GPUs of NVIDIA V100 and Google TPU can execute operations faster using FP16 than FP32. In NVIDIA's GPUs, FP32 data is converted to FP16 data

in SMs, then fed into the tensor core for accelerated operations through a specific FP32-FP16 data workflow.

3.4 LightSeq Training

Apart from the computationally intensive operations in the Transformer Encoder, other operations such as transpose, bias addition, softmax, and data transfer between the device and host may also impact performance. To enhance the fine-tuning process, it is of interest to minimize the potential overhead caused by these operations.

The current implementation of the BERT model by Hugging Face prioritizes adhering to the original structure of the model rather than focusing on performance optimization. As a result, Hugging Face's implementation provides a stable and convenient API interface for data scientists to use, however there are many optimizations that can be done to increase performance. Table 3.3 shows the average execution time for Softmax and LayerNorm kernel in original Hugging Face implementation. Furthermore, the time consumption of encoder implemented Hugging Face in both the forward as well as backward without mixed precision is also found in the table.

Kernel	Average Duration
softmax_warp_forward	497 μs
softmax_warp_backward	712 μs
vectorized_layer_norm_kernel	80 μs
encoder forward	14.8 ms
encoder backward	25 ms

Table 3.3: Average execution time of common kernels within Hugging Face model.

LightSeq offers an alternative implementation of the Hugging Face Transformer, which surpasses Hugging Face's BERT model implementation in terms of performance. This is accomplished by sacrificing some of the model's flexibility and eliminating redundancy overhead in the existing implementation, while retaining the existing API interface.

To achieve this objective, LightSeq utilizes a number of strategies, such as fusing Bias addition with Q, K, and V matrix reshaping to reduce the number of kernel starts required. Additionally, LightSeq optimizes the Softmax function by utilizing different kernels to compute matrices of varying sizes. In order to further improve performance, LightSeq allocates a limited number of registers to hold frequently accessed variables. This approach is informed by the size of the matrices being processed, thereby minimizing the memory footprint required for variable storage. Moreover, LightSeq leverages the hardware architecture to reduce the overhead of exchanging information between warps. To accomplish this, LightSeq incorporates

a reduce operation within warp that utilizes the shfl_xor_sync primitive to enable efficient intra-warp information exchange.

In this thesis, LightSeq is implemented using the Hugging Face BERT model by replacing the Hugging Face BERT model's encoder-layers with LightSeq's optimized Transformer encoder-layers. The training process will then be benchmarked and profiled to analyze the reduction in time and energy consumption compared to the original Hugging Face model.

The process of replacing the Hugging Face encoder layers is simple, as demonstrated in the following code:

```
from ls_module import LSBertForSequenceClassification

def from_pretrained(self, *args, training_args, model_args, ...):
    self.config = kwargs["config"]
    model = super().from_pretrained(*args, **kwargs)
    if model_args.module_type == 1 or model_args.module_type == 2:
        inject_ls_layer(model, training_args,
            model_args, self.config)
    return model

model = LSBertForSequenceClassification.from_pretrained(
        model_args.model_name_or_path,
        training_args=train_args,
        model_args=model_args,
        config=config)
```

This type of replacement allows one to still use the model as if it were an Hugging Face model, but with the LightSeq encoder layers instead, meaning that any Hugging Face Trainer, or other framework previously used will still work.

The LightSeq optimized model will be used if the model type is 1 or 2 (Type 1 means use standard lightseq encoder, type 2 means enable lightseq quantization, type 2 is not used in this thesis); otherwise, the Hugging Face model is used. To implement this a LightSeq Encoder class using the **inject_ls_layer** function is created. The pre-trained parameters from the Hugging Face encoder will be copied into the LightSeq encoder instance, and then the replacement will be completed. This process ensures that a switch between the two models seamlessly based on the specified model type can occur.

3.5 Distributed Training

The current workflow prioritizes improving performance through hardware optimization techniques. However, a significant challenge in deep learning is memory constraints that can be overcome by using distributed training. Therefore, the next step in optimizing performance involves implementing data parallelism for faster

training. This widely used method is scalable and enhances performance without negatively affecting model performance. In essence, distributed training with multiple devices can be equated with training with a larger batch size, potentially leading to improved model generalization.

3.5.1 Enabling Data Parallel

Distributed training with parallel dependency data can be performed using the accelerator library interface. The distribution parameters can be automatically extracted from a YAML configuration file. Therefore, the required number of GPUs only needs to be specified in the YAML file. The NVIDIA NCCL library is commonly used as the communication backend, which provides efficient communication between GPUs.

```yaml
compute_environment: LOCAL_MACHINE
distributed_type: MULTI_GPU
machine_rank: 0
main_process_ip: null
main_process_port: null
main_training_function: main
mixed_precision: fp16
num_machines: 1
num_processes: 2 # Using two GPU for training
use_cpu: false
```

The accelerator the parameters required for distributed data parallelism are passed in as described in the code below.

```python
from accelerate import DistributedDataParallelKwargs

ddp_kwargs = DistributedDataParallelKwargs()
accelerator = Accelerator(kwargs_handlers=[ddp_kwargs])
```

The accelerator can automatically deploy a distributed training environment and establish communication between devices. In the trainer, the accelerator also automatically calls the backward method to perform an AllReduce Sum operation on the loss across all devices. During this process, each device accumulates gradients for all samples. Finally, the accumulated gradients are divided by the total number of samples to obtain the required gradient update for the current training batch.

3.5.2 Backpropagation Time Estimation

During the process of distributed training, a significant amount of communication between devices is observed to occur during the backpropagation phase, with the duration of each training step being significantly correlated with communication time. In this subsection a method to estimate the backpropagation time in the case of data parallelism. Definitions in Table 3.4.

To estimate the time required to backpropagate the entire neural network, the time

Notion	Description
T_i^{start}	Start time of i^{th} task
$T_i^{previous}$	Time point when $(i-1)^{th}$ task finishes
$T_i^{compute}$	Time point when i^{th} task completes computation
T_i^{done}	Time point when i^{th} task is finished
$D_i^{compute}$	Duration of computation for i^{th} task
D_i^{comm}	Duration of communication for i^{th} task
$Overhead$	Overhead between two tasks

Table 3.4: Notions used for Backpropgation time estimation

required for each layer during backpropagation is predicted and the communication time between GPUs. Two scenarios which are shown in Figure 3.5: when the communication channel is available immediately after completing the current task computation and when it is not. In the former case, the transmission is assumed to start immediately and approximates the task completion time as the sum of computational and communication time.

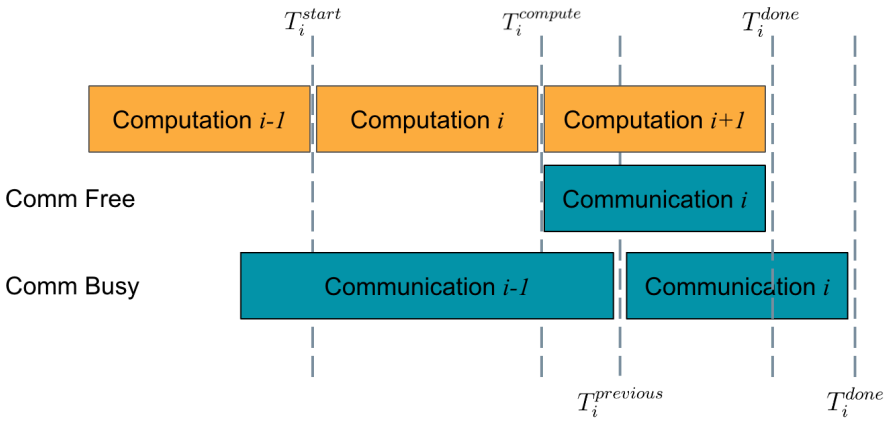

Figure 3.5: Backpropagation workflow in distributed training.

However, if the communication is occupied then the communication time of previous task should be considered. This is a complex situation because there may be multiple communication requests from different tasks piled up and not processed, making it difficult to estimate the delay time for the communication of the current task.

The approach to calculating the overall backpropagation time is iterative. By computing the model in the order of each layer of the network stack and estimate computation and communication time for each layer. The assumption is that the start time of backpropagation is zero and the communication channel is idle, which could be written as $T_0^{start} = 0$, $T_0^{previous} = 0$. As well as the assumption that the

next layer of the backpropagation task starts immediately after the computation of the current layer is completed.

To calculate the completion time of the current task certain rules are established. First determine $T_i^{compute}$ and set the start time of the next layer of network backpropagation to $T_{i+1}^{start} = T_i^{compute} + Overhead$. Next, calculate the current task end time T_i^{done} immediately after that. If the communication channel is available then use $T_i^{compute}$ to determine T_i^{done}. If the communication channel is not available instead use $T_i^{previous}$ to calculate T_i^{done}. Finally set $T_{i+1}^{previous} = T_i^{done}$.

```
for layer in layer_list:
    if is last layer:
        print("Backpropagation time:", layer.T_previous)
        break
    layer.T_compute = layer.T_start + layer.D_compute
    next_layer.T_start = layer.T_compute + overhead
    if layer.T_previous > layer.T_compute:
        # wait previous finish
        layer.T_done = layer.T_previous + layer.D_communicate
    else:
        layer.T_done = layer.T_compute + layer.D_communicate
    next_layer.T_previous = layer.T_done
```

3.6 Training on a TPU

The previous optimizations performed have focused on accelerating training when using CUDA-based architecture, such as the NVIDIA V100 or A100. However, as previously highlighted in Section 2.3.2 the TPU architecture can also offer high performance training, with high throughput and low energy consumption. It is therefore of interest to compare the performances of GPUs and TPUs, to evaluate baseline performance but also the optimized GPU performance relative to a dedicated AI accelerator such as TPUs.

It is important to note that whilst one of the primary focuses has been on evaluating energy consumption, this is not possible when using TPUs on either Google Colab or Kaggle. This is due to Google not providing any tools the likes of NVIDIAs System Management Interface (SMI), therefore the energy consumption of TPUs will not be explored when benchmarking. The main areas of focus when comparing TPUs and GPUs are therefore Time and Cost.

3.6.1 Training Frameworks

Implementing the previously used workflow (Text Classification with BERT) can be done using two different frameworks, PyTorch XLA or TensorFlow. The former is a modified version of PyTorch which was previously used for the GPUs, it allows for converting PyTorch-based models and workflows to be used on TPUs. Allowing

re-usage of the previously developed workflow with minor changes. The latter is a framework developed by Google for deep learning. Its important to note that TPUs are specifically tailored to use TensorFlow so there may be a difference in performance.

The Hugging Face Transformer and Datasets libraries can be used, regardless of which framework is chosen, because Hugging Face implements both. The use of the BERT base model, tokenizer, and IMDb dataset, which have been used previously, is permitted. However, the Hugging Face Accelerate library can only be used with PyTorch XLA.

The implementation of this workflow in PyTorch XLA and TensorFlow is done to evaluate the potential performance differences between the two frameworks. The PyTorch XLA version uses the Hugging Face acceleration library to help allocate the training process between different TPU cores, while the TensorFlow version is handled automatically.

3.7 Systematic Benchmark

Different optimization techniques were combined by us to examine the performance of various optimization combinations on a range of different hardware, including Nvidia GPUs and Google TPUs. The method used involves combining all the optimizations and executing each combination on several different devices. The parameters that are modified are based on previous optimization techniques that were used and developed in the first few sections of the method chapter. For a detailed overview, see the list in Table 3.5. Key indicators for this systematic benchmark evaluation are monetary cost, time, and energy, although energy will not be considered on the TPU due to limitations.

Parameters	Content
Optimizer	AdamW (unfused multi), Adan (fused multi, GPU)
Mixed Precision	FP32, FP16, BF16(TPU)
Encoder type	Hugging Face, LightSeq(GPU)
Batch size	GPU(8, 16, 32), TPU(64, 128, 256, 512, 1024)
Hardware	V100, A100, T4, A10, TPUv2, TPUv3

Table 3.5: Showing the parameters used for the different hardware in the systematic benchmark.

3.7.1 Hardware examined

The benchmark test will examine various NVIDIA GPUs provided by Microsoft Azure, including A100, V100, A10, and T4. Additionally, TPUv2 and TPUv3 will also be tested, both of which are available on Google Cloud. The specifications for the selected hardware can be found in Table 3.6.

Hardware	Clock Frequency	Memory and Bandwidth	TDP	Release Date
T4	585MHz	16GB, 320GB/s	70W	2018
V100	1370MHz	16GB, 1750GB/s	250W	2017
A10	1695MHz	24GB, 600GB/s	150W	2021
A100	1512MHz	80GB, 1935GB/s	300W	2021
TPUv2	700MHz	16GB, 600GB/s	280W	2017
TPUv3	940MHz	32GB, 900GB/s	220W	2018

Table 3.6: Showcasing the hardware specifications of the selected hardware. Important to note is that Clock Frequency when referring to GPUs is the CUDA core clock frequency.

The main competitors are A100 and TPUv3, as well as V100 and TPUv2, given their similar release dates. A10 and T4 are mainly used for inference, but in this thesis they will be used for training, due to their low cost relative to the other cards.

4

Results

This chapter highlights the results from the various experiments conducted throughout the thesis, linking back to the previous chapter.

4.1 Experimental Setup

This subsection aims to show and explain the experimental setup used when conducting the benchmarks presented in the upcoming sections. Briefly describing the model, platforms, datasets, training parameters and metrics used in the remainder of the result section.

4.1.1 Models

In this thesis work, the focus is mainly on fine-tuning a natural language processing task with the BERT-base pretrained model provided by Hugging Face. BERT-base model has a stack of 12 Transformer encoders, the number of multi-headed attention layers is 8, and the size of the hidden layer is 784, with a total number of parameters of about 110M.

4.1.2 Platform

The software ecosystem used in this thesis consists of the hugging face BERT-base model [4], PyTorch 1.12.1, CUDA 11.3, and Python 3.9. In order to perform TPU fine-tuning, we have incorporated TensorFlow 2.11 and the PyTorch XLA library to modify our workflow.

Our computing infrastructure boasts an Azure-hosted Databricks node equipped with a 6-core CPU and NVIDIA V100 PCIe GPU featuring 16GB of video memory. These components utilize the PCIe bus and possess a collective RAM capacity of 48GB per node. Furthermore, we possess an A100 node that leverages an NVIDIA A100 PCIe GPU which showcases 40GB of video memory. For TPU computing, we rely on the 8-core TPUv2 platform provided by Google Colab.

4.1.3 Dataset

The IMDb dataset [36] is a collection of information about movies and TV shows available on the Internet Movie Database (IMDb), a popular online database con-

taining information on movies, TV programs, video games, and streaming content. This dataset includes a variety of attributes such as title, release year, genre, rating, runtime, cast and crew information, plot summaries, user reviews, and other related metadata. It is publicly accessible and commonly used in research studies focused on movie and television recommendation systems, sentiment analysis, and natural language processing. Consequently, it is a valuable resource for those interested in examining the entertainment industry, movie ratings, and user behavior. IMDb provides access to data via APIs, flat files, and other methods.

One of the primary applications of this dataset is sentiment analysis, which aims to identify the reviewer's attitude to a movie as either positive or negative based on the content of the movie review. The dataset contains a total of 50,000 reviews, which are divided into two sets of 25,000 for training and testing purposes. Each review is assigned a label of either 0 or 1, with 0 indicating a negative sentiment and 1 indicating a positive sentiment. In this thesis the IMDb dataset is used to fine-tune on the BERT pre-trained model so that it can accomplish the task of sentiment classification of movie reviews. To be noted however is that the focus in the thesis isn't on this specific task, but on the fine-tuning process.

In all our tests, the same random seeds are used, so the split of the training set as well as the test set is kept consistent.

4.1.4 Training Settings

For the Fused Adan test, the Adan optimizer used parameters $\beta_1 = 0.98$, $\beta_2 = 0.92$, $\beta_3 = 0.99$, $lr = 1 \cdot 10^{-4}$, $wd = 0.01$. All hyperparameters are set using the values recommended by the original paper, in addition the batch size is 16 for each device.

In Section Fused Adan, the goal was to evaluate and compare the unfused and fused Adan kernel. To achieve this, the stopping criterion was set to fine-tune the full 3 epochs. The different kernel implementations for Adan are also investigated, conducted using a 1-step benchmark, which means the training loop will break after only a singular training step.

The same stop criterion is used for Mixed Precision Training, LightSeq, Distributed Training, and A100/TPU fine tuning scenario. Specifically, the fine tuning is stopped after three epochs. The training time in all tables is obtained by subtracting the total time from the time consumed for validation at each epoch, a method that highlights the actual time consumed for training. Further details on the benchmark settings can be found in Table 4.1.

Benchmark	Hardware	Optimizer	Stop criterion
1	NVIDIA V100	Adan(f_m&uf_s)	3 Epochs
2	NVIDIA V100	Adan	One-step
3	NVIDIA T4	Adan	One-step
4	NVIDIA V100	Adan	3 Epochs
5	NVIDIA V100	Adan	3 Epochs
6	NVIDIA V100 1/2/4	Adan	3 Epochs
7	Google TPUv2	AdamW(uf_m)	3 Epochs
8	NVIDIA V100/A100	AdamW(uf_m), Adan(f_m)	3 Epochs

Table 4.1: Benchmark settings, f_m means fused multi tensor, uf_s means unfused single tensor, etc.

4.1.5 Metrics

Training Time: Refer to the training time criterion provided by MLperf training [37]: the clock time when a model is trained on a specific dataset to reach the quality target. This metric will be defined as **clock time consumed by the system, denoted as T**, the shorter the training time, the better performance.

Cost: If an optimization can yield lower costs, then it is of consideration. Likewise, if a optimization leads to a vastly higher cost then it would require changes, or be dropped from consideration. Volvo offers the Azure computing platform with a single NVIDIA V100 GPU for about 5 DBU (0.55$ per DBU) per hour, so the idea is to reduce cost as much as possible by reducing time consumption. This metric will be defined as **total cost for training model, denoted as C**, the less the training cost, the better performance.

Energy Consumption: With the usual focus being on improving accuracy first-hand in machine learning, inefficiencies such as high energy consumption sneak in. Therefore it would be of interest to measure the energy consumption of the fine-tuning process being accelerated, to see if improvements can be made in reducing energy usage, whilst increasing or maintaining performance. This metric will be defined as **total energy consumption for training model, denoted as E**, the less the training energy cost, the better performance. In this thesis we use kWh as the unit, 1 kWh equals about 3600000 joules.

Achieved Occupancy (Only compatible with NVIDIA GPU): For most cases such as memory bandwidth bound kernels, a higher value often translates to better performance, especially when the initial value is very low. Occupancy is the ratio of active warps on a streaming multiprocessor (SM) to the maximum number of active warps supported by the SM. The theoretical occupancy of a kernel is the upper limit occupancy of this kernel, limited by multiple factors such as kernel shape, kernel used resource, and the GPU compute capability. This metric will be defined as **average occupancy during training model, denoted as O**, the higher the training achieved occupancy, the better performance.

SM Throughput (Only compatible with NVIDIA GPU): The computational throughput of a streaming multiprocessor (SM) can be determined by measuring the percentage of the theoretical maximum computational throughput. This metric is valuable for analyzing kernel behavior and observing the impact of various kernels and the number of startup threads on throughput. Understanding this metric can provide insight into the optimal configuration for maximizing computational efficiency.

Memory Throughput (Only compatible with NVIDIA GPU): Memory throughput is the ratio of the actual bandwidth utilized on the current device to the theoretical maximum bandwidth.

4.2 Fused Adan

4.2.1 Benchmark on Training process

The purpose of Benchmark 1 was to evaluate the training performance of unoptimized Adan kernel, which is unfused with single tensor access Adan, and optimized Adan kernel, which is fused with multi tensor access Adan. This benchmark is conducted with 3 full epochs training with same hyper parameters.

Optimizer	Time/s	Cost/$	Energy/$kW \cdot h$	Occupancy	Test Accuracy
Unfused Adan	2286.85	1.75	0.1425	48.66%	93%
Fused Adan	2034.45	1.56	0.1243	46.64%	94%

Table 4.2: Benchmark 1, Comparison between different optimizer.

Table 4.2 presents the results obtained, which indicate that the fused Adan optimizer with multi tensor access reduces training time, energy consumption, and training costs. However, there is a slight decline in hardware utilization. Furthermore, the trend of validation loss and accuracy during the fine-tuning process was also analyzed, as shown in Figure 4.1. Moreover, the results indicate that the adoption of fused Adan does not contribute to a reduction in validation loss or an improvement in accuracy. There is no observable difference in computational stability between fused Adan and Adan's native PyTorch implementation.

Overall, our results suggest that the fused Adan kernel is more efficient than unfused Adan in terms of training time, energy consumption, and training costs, while maintaining comparable accuracy and computational stability.

4.2.2 Benchmark on Optimization process

Benchmark 2 examines Adan's hardware performance with respect to tensor access and operator fusion. The test uses the fine-tuning task on the IMDb dataset but with a one-step truncated training process to speed up benchmark. This benchmark acquires data from the model's forward and backward processes, along with the

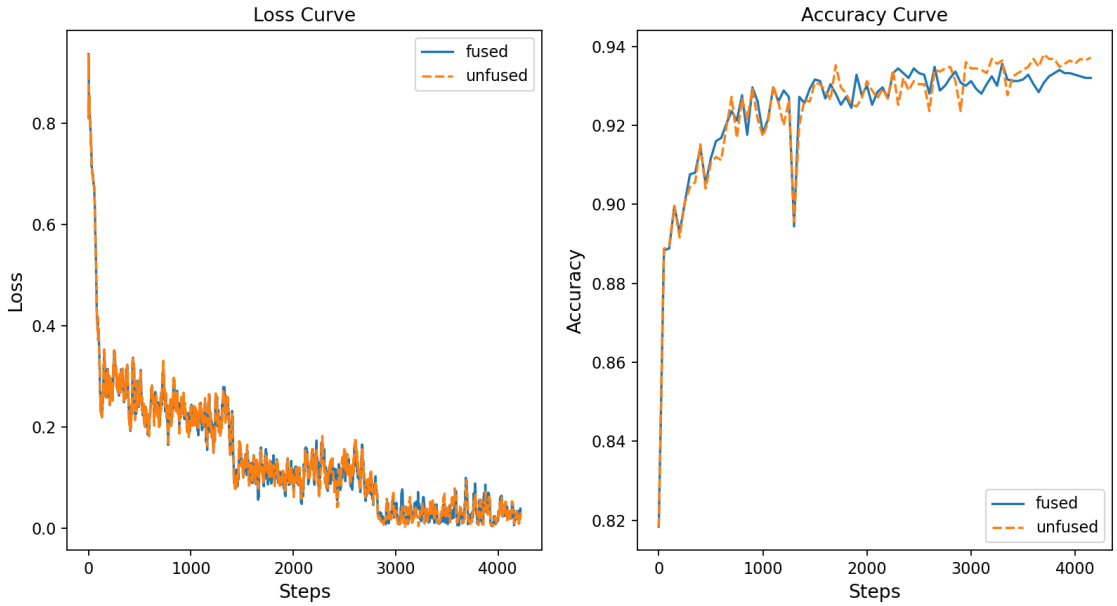

Figure 4.1: Benchmark 1, Loss and accuracy for unfused and fused Adan.

parameter update process, with a primary focus on the optimizer's involvement in parameter update.

Workload	Duration/ms	Computation/ms	Preparation/ms
IMDb_adan_single_unfused	232.866	169.331	63.535
IMDb_adan_single_fused	96.05	32.566	63.484
IMDb_adan_multi_unfused	83.486	24.591	58.895
IMDb_adan_multi_fused	71.741	9.473	62.268

Table 4.3: Benchmark 2, One-step benchmark time results

Table 4.3 above provides information on the number of GPU kernel launches and the duration of the step during the execution of optimizer.step(). The time spent on these kernel launches consists of two components: the time spent preparing the data before the computation and the time spent on the actual computation. Based on the table, it is evident that the data preparation time is relatively constant across all four tests, averaging about 60ms. Conversely, the computation time varies significantly.

Workload	Total	Computation	Preparation	Energy/mJ
IMDb_adan_single_unfused	4422	3618	804	6177
IMDb_adan_single_fused	1407	603	804	3929
IMDb_adan_multi_unfused	912	108	804	2784
IMDb_adan_multi_fused	826	22	804	2429

Table 4.4: Benchmark 2, Different Adan kernel one-step benchmark results

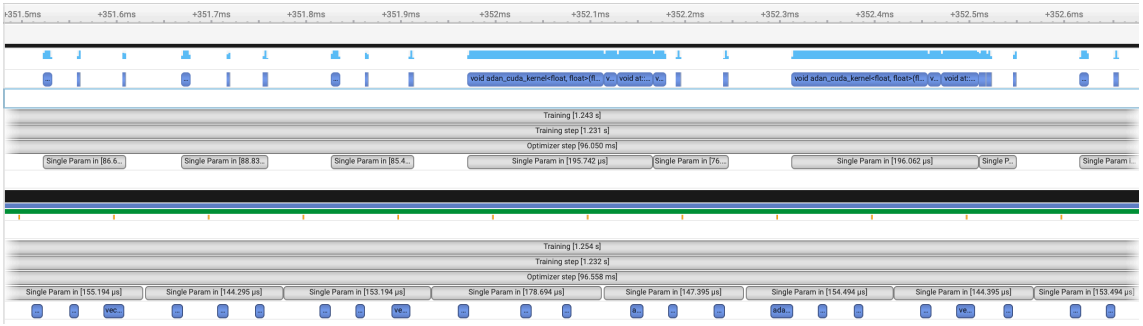

Figure 4.2: Benchmark 2, Fused Adan kernel with single tensor access launch time graph, showing kernel launch is very loosely distributed over the timeline

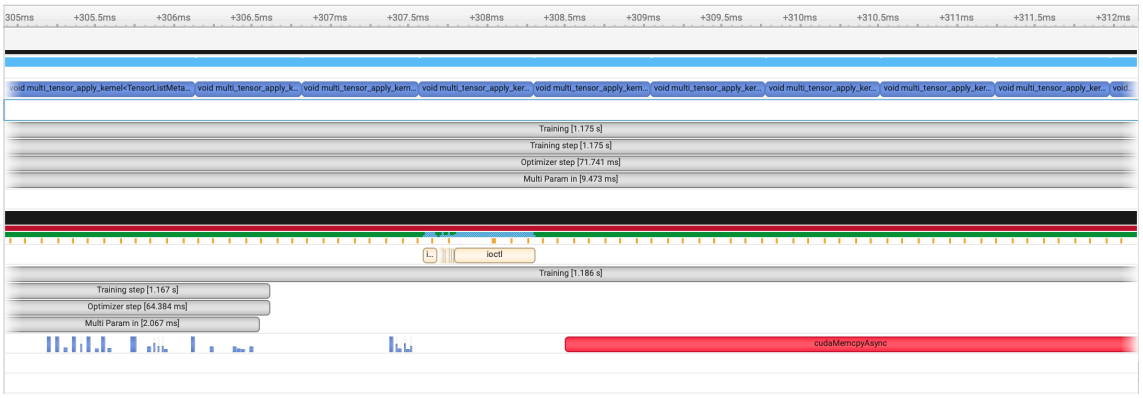

Figure 4.3: Benchmark 2, Fused Adan kernel with multi tensor access launch time graph, showing kernel launch is compactly distributed over the timeline

In Benchmark 2, the number of kernels and energy consumption used for computation and data preparation was investigated, results are shown in Table 4.4. As shown, the number of kernels used for data preparation is comparable to the number of kernels used for computation. This finding is consistent with the observed performance of the time used for data preparation. Our results shows that using multi tensor access and operator fusion can significantly reduce GPU kernel launch overhead and speed program execution.

When examining energy consumption, it is evident that a decrease in the number of kernels corresponds to a decrease in energy consumption. This observation implies that reducing the number of kernels used in a computation leads to a proportional decrease in both runtime and energy consumption. Thus, using fewer kernels can effectively reduce both computational time and energy usage.

In addition, our results shows that multi tensor access can minimize GPU idle time. Figure 4.2 and Figure 4.3, demonstrate that when using fused single tensor access, there are many idle periods between GPU kernels that cannot be used. However, when using fused multi tensor access, GPU kernels are tightly arranged, resulting in minimal GPU idle time.

In Chapter 3, it was illustrated that the Adan kernel's active time during the one-step benchmark, without kernel fusion and multi-tensor access optimization, is only

18.48% of the kernel's running time. The following content presents the active time ratio of the compute unit for kernel fusion and multi-tensor access optimization separately, as well as a combination of the two methods. The aim is to provide a clearer understanding of the performance impact of these optimization techniques.

Workload	Running time/μs	Active time/μs	Active ratio
Adan_single_unfused	169331	31293.63	18.48%
Adan_single_fused	32566	10981.64	33.72%
Adan_multi_unfused	24591	24504.25	99.65%
Adan_multi_fused	9473	9455.02	99.81%

Table 4.5: Active time ratio for different Adan kernel implementations

Table 4.5 demonstrates a significant increase in the active ratio, from 18.48% to 99.81%, as a result of implementing kernel fusion and multi tensor access.

Moreover, using kernel fusion with single tensor access enhances the active ratio by approximately 15.24%. On the other hand, using kernel fusion with multi tensor access only increases the active ratio by 0.16%. Thus, it is evident that kernel fusion can yield superior outcomes when dealing with a large number of kernels.

The following subsection will delve into the reasons behind the significant impact of multi tensor access.

4.2.3 Benchmark on Adan fused kernel

Benchmark 3 compares the performance of three different kernels (i.e., Kernel A, Kernel B, and Kernel C) in fused Adan. Specifically, Kernel A was optimized for single tensor access using vector instruction acceleration, Kernel B used more threads, and Kernel C incorporated loop unrolling acceleration for multi-tensor access. Each kernel was evaluated by using the same benchmark settings as in the previous benchmark as well as by conducting a single training step.

Since tensor access differs in each kernel, the number of threads launched in each kernel varies slightly, which may affect hardware utilization. To ensure a fair comparison the first tensor accessed by the kernel at startup was selected. The size of the tensor accessed by Kernel A and Kernel B was the same, but larger than that accessed by Kernel C, whose size was determined by chunk size using NVIDIA Apex's recommended chunk size.

Our results show that Kernel A, which used vector instructions, had significantly lower LSU utilization than Kernel B and Kernel C. This observation suggested that using vector Load/Store instructions could reduce LSU utilization, although it could lead to longer LSU usage, resulting in stalling of other warps that require LSU usage.

Kernel B used fewer registers to accommodate the large number of launch threads, which could reduce the number of active warps on the SM, hindering the ability to hide access latency through warp switches. As a result, Kernel B exhibited a higher

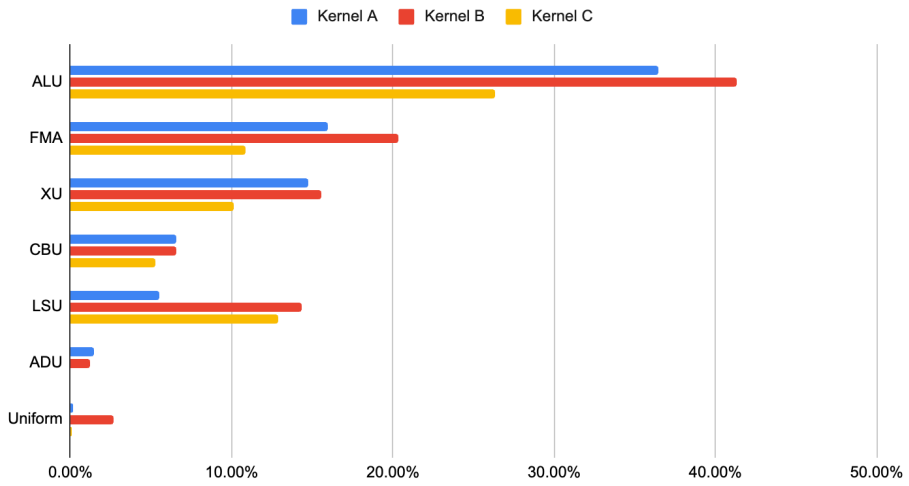

Figure 4.4: Benchmark 3, GPU pipeline utilization

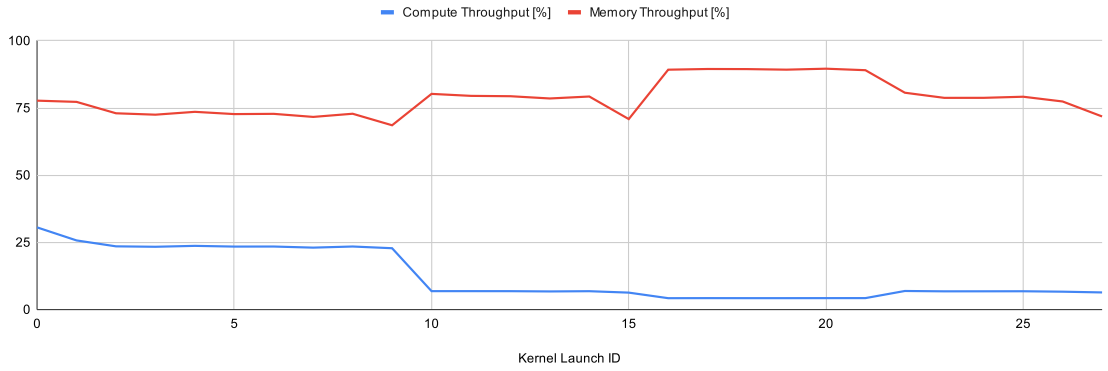

Figure 4.5: Figure shows the SM and memory throughput of the GPU when using Multi Tensor Access in one-step benchmark.

frequency of LDG instructions than Kernel A and Kernel C, which could negatively impact performance.

Kernel C accessed a smaller tensor size than Kernel A and Kernel B, which could affect pipeline utilization. However, multi-tensor access maintained a stable number of threads started by the kernel, improving overall utilization efficiency during the computing process. The throughput of various kernel sizes when accessing different tensors is presented in Table 4.6.

In multi-tensor access, Kernel C exhibited consistent SM and memory throughput. This finding suggests that controlling the number of threads deployed during kernel launch within an optimal range can significantly affect hardware utilization and throughput. Therefore, it is recommended to use multi-tensor access and maintain an appropriate number of threads to maximize performance efficiency.

Kernel	Grid Size	Block Size	SM Throughput	Memory Throughput
Kernel A	21747	1024	39.45%	82.44%
Kernel A	2304	1024	37.34%	79.22%
Kernel A	576	1024	31.44%	62.96%
Kernel A	2	1024	0.78%	0.89%
Kernel A	1	1024	0.45%	0.50%
Kernel B	21747	1024	47.09%	70.67%
Kernel B	2304	1024	43.3%	72.58%
Kernel B	576	1024	40.49%	64.35%
Kernel B	2	1024	1.61%	1.53%
Kernel B	1	1024	0.82%	0.84%
Kernel C	320	512	30.57%	77.73%
Kernel C	177	512	23.52%	73.03%
Kernel C	52	512	22.82%	68.57%

Table 4.6: Benchmark 3, Typical kernel size and throughput for different Adan kernel implementation

4.3 Mixed Precision Training

This section of the benchmark test assesses the impact of mixed precision training on model training. To prevent extraneous interference in the training process, the fused adan optimizer is used.

Training Settings	Time/s	Cost/$	Energy/$kW \cdot h$	Occupancy	Test Accuracy
FP32 training	2034.45	1.56	0.1243	46.64%	94%
FP16 training	813.34	0.73	0.0425	58.4%	93%

Table 4.7: Benchmark 4, Mixed precision training result

Figure 4.6 shows the behavior of validation loss and accuracy throughout the training process, both with and without mixed precision training. The incorporation of FP16 data does not have a significant impact on the training process.

Using mixed precision training results in significant reductions in training time and energy consumption by 60% and 66%, respectively, while simultaneously increasing hardware utilization by 11.76%. This gain is primarily due to NVIDIA V100's dedicated Tensor Core, which efficiently processes FP16 data for computationally intensive tasks. In addition, the Tensor core functions as a dedicated acceleration unit specifically designed for FP32-FP16 operations, contributing to the observed energy benefits. The memory usage was also observed. The peak memory usage reached 12623.5 MB without mixed precision training, while the peak memory usage decreased to 10082.1 MB with mixed precision training.

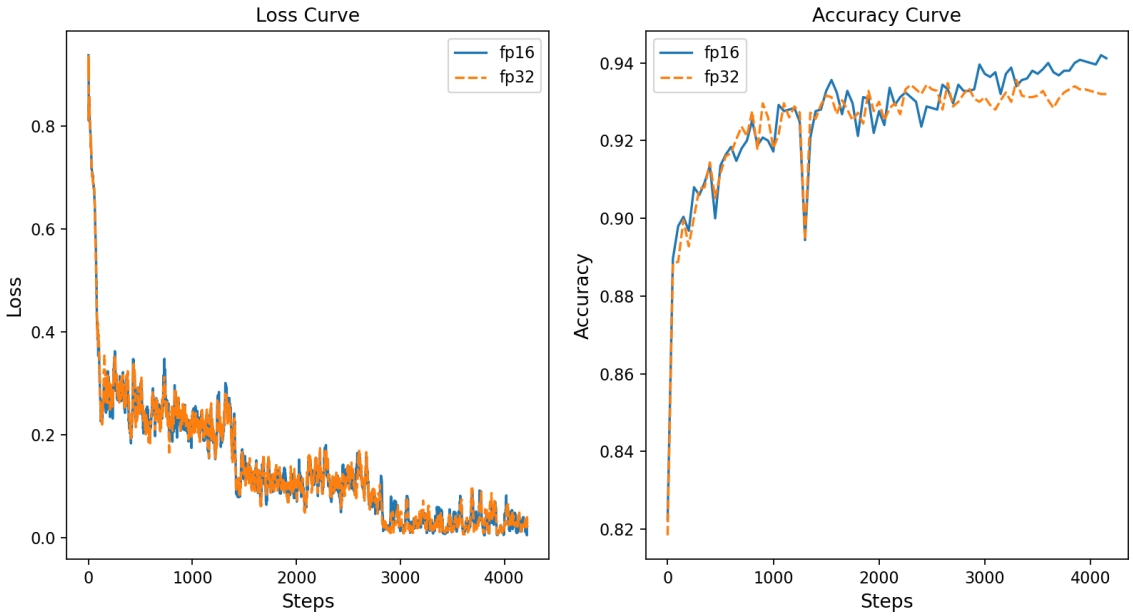

Figure 4.6: Benchmark 4, Mixed precision training loss and accuracy

Mixed precision training provides a greater advantage in terms of energy consumption (about 5%) compared to time consumption. Therefore, to train the model using less energy, a similar dedicated computing module should be considered to speed up the training and reduce energy loss.

According to the findings depicted in Figure 4.7, the utilization of mixed precision has resulted in a notable reduction of the proportion of computationally demanding sgemm operators in the total elapsed time. Specifically, the proportion reduced from 93.6% to 37.1%. This outcome is indicative of the accelerated computation speed of the sgemm kernel when mixed precision training is employed. For instance, the typical execution time for a tensor core accelerated matrix sgemm kernel named 'volta_s884gemm_fp16_64x128_tn' is 230 μs, whereas the execution time for a similar matrix sgemm kernel named 'volta_sgemm_128x64_nt', without mixed precision, is 1274 μs.

4.4 LightSeq Training

This section show the results from training using the standard implementation in HuggingFace and the acceleration library LightSeq.

Training Settings	Time/s	Cost/$	Energy/$kW \cdot h$	Occupancy	Test Accuracy
FP32 HuggingFace	2034.45	1.56	0.1243	46.64%	94%
FP32 LightSeq	1861.89	1.42	0.1197	42.0%	92%
FP16 HuggingFace	813.34	0.62	0.0425	58.4%	93%
FP16 LightSeq	535.45	0.41	0.0325	41.48%	92%

Table 4.8: Benchmark 5, Training using Hugging Face and LightSeq

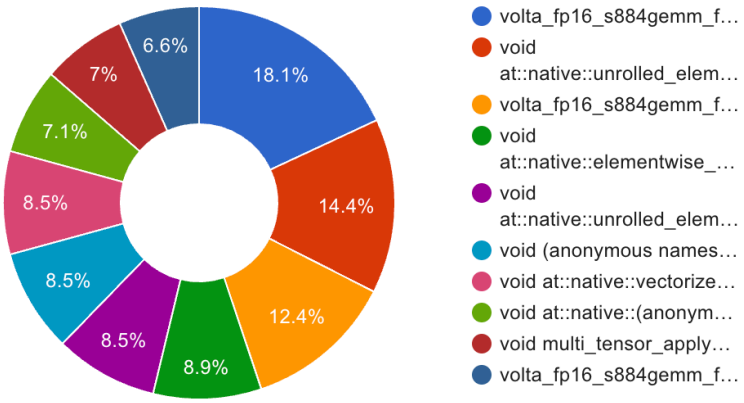

Figure 4.7: Figure shows the percentages of the top 10 GPU kernels with the highest time consumption during one training step using Hugging Face model with mixed precision.

Table 4.8 show that LightSeq can provide a decent speed-up when fine-tuning the BERT model relative to the HuggingFace implementation. When training without mixed-precision (FP32) the speed-up using LightSeq is moderate, around 9% faster time-to-train with the energy consumption decreasing accordingly. However, HuggingFace does show a higher test accuracy with 94% compared to LightSeq's 92% which does indicate that some of the optimization techniques impacts accuracy albeit minimally. Comparing LightSeq with HuggingFace when fine-tuning using mixed-precision (FP16) show a significant speed-up when using the LightSeq library. LightSeq's time-to-train is around 52% faster than that of HuggingFace while being around 30% more energy efficient. Moreover, the test accuracy remains the same at 92% for LightSeq whilst a very small drop occurs for HuggingFace at 93%.

The results shown in Table 4.8 does show that the optimization techniques employed in LightSeq minimizes the impact on accuracy loss whilst providing a speed-up. Especially when using mixed-precision, which was a main focus when the framework was built. Furthermore, Figure 4.10 show that using LightSeq does lead to a slightly higher validation loss compared to HuggingFace's implementation. Likewise, the validation accuracy of the LightSeq implementation is lower or equal to that of HuggingFace.

Figures 4.8 and 4.9 demonstrates the results of a singular training step using FP16 set-ups for both HuggingFace and LightSeq. The purple graphs show the launched kernels by both set-ups, whilst the blue displays utilization. The figures highlight the impact of LightSeq's fused kernels, which reduces the sporadic kernel launches seen in Figure 4.8 which decreases GPU idling time, and increases the performance.

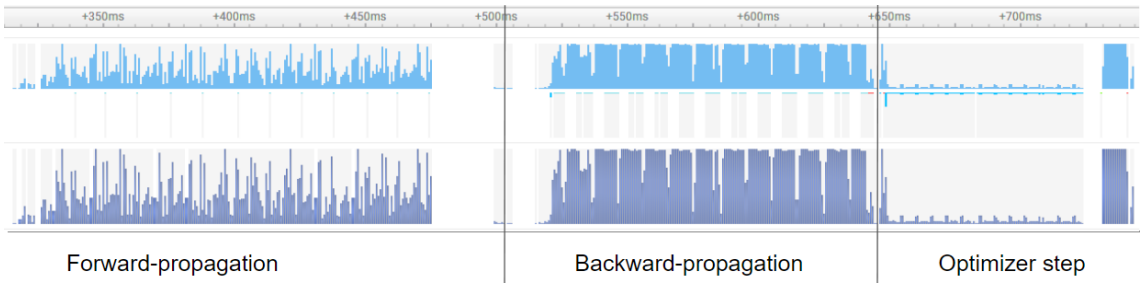

Figure 4.8: Figure shows the hardware utilization when training the Hugging face model using mixed precision in the one-step benchmark.

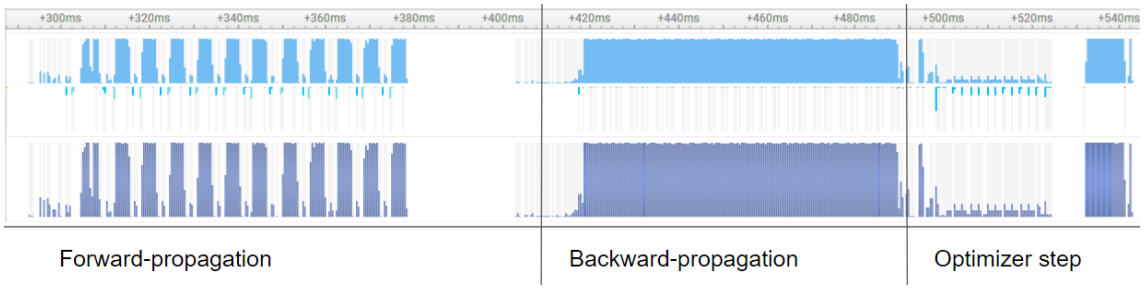

Figure 4.9: Figure shows the utilization of the hardware in the one-step benchmark when training the LightSeq model using mixed precision.

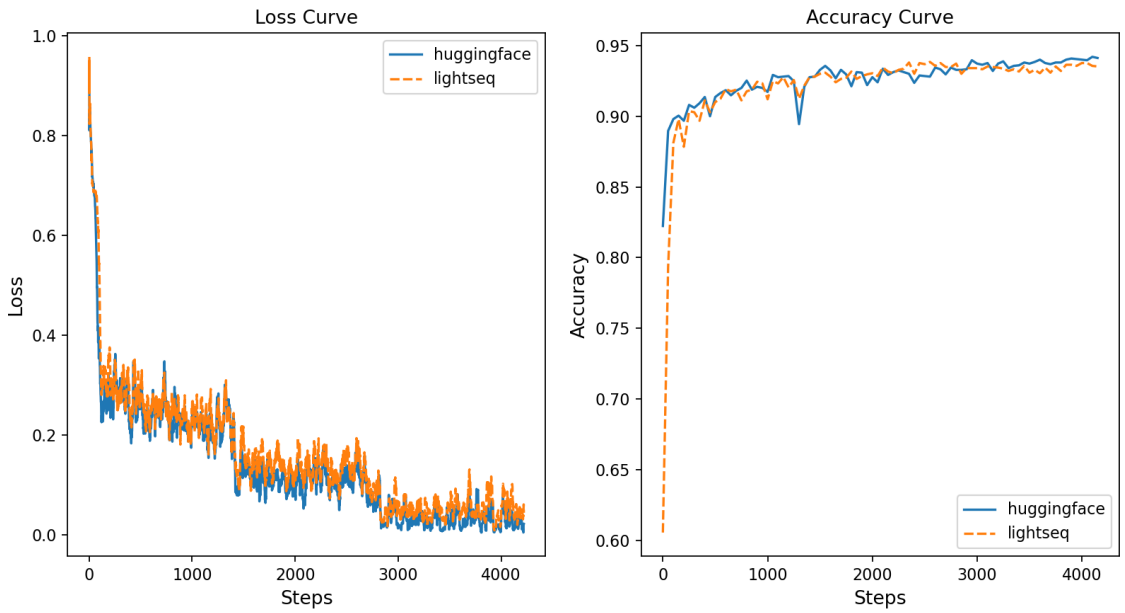

Figure 4.10: Benchmark 5, FP16 Training loss and accuracy

4.5 Distributed Training

4.5.1 Data Parallel

In this subsection, we explore the use of data parallelism as a means of distributed training, which is considered the easiest approach. We apply this method to train the entire IMDb dataset for three full epochs, using 1, 2, and 4 GPUs for training, evaluated optimized LightSeq encoder module performance with enable mixed precision training.

Our goal is to examine the performance acceleration achieved with different GPU configurations. In theory, an increase in the number of GPUs should result in a linear reduction in training time without considering any additional overhead that may result from distributed training. However, in practice, due to inter-GPU communication, we were unable to achieve a linear acceleration ratio.

We also aim to investigate GPU energy consumption during distributed training. In general, estimating overall energy consumption is difficult due to the reduction in training time and the increase in the number of GPUs used. Therefore, we could discuss the distribution of energy consumption among multiple GPUs in a data parallel environment based on our experimental results.

Settings	Time/s	Cost/$	Energy/$kW \cdot h$	Occupancy	Test Accuracy
1 GPU ls	535.45	0.41	0.0325	41.48%	92%
2 GPU ls	457.55	0.70	0.0389	22.37%	95%
4 GPU ls	281.41	0.86	0.0441	16.43%	93%

Table 4.9: Benchmark 6, Data parallel training result

Following discussion of the effects of distributed training on a range of metrics is conducted on the training data of the BERT model optimized by LightSeq.

Table 4.9 shows the results of benchmark 6, indicating that training time does not decrease proportionally. This trend highlights the impact of communication overhead on the acceleration of distributed training. Specifically, the speedup ratio is 1.17 and 1.9 with 2 GPUs and 4 GPUs, respectively. From the time perspective, training time decreased by 14.5% using 2 GPUs for training; 47.4% using 4 GPUs for training. From the energy perspective, the energy consumption for training with 2 GPU increased by 19.7%; the energy consumption for training with 4 GPU increased by 35.5%.

Our results indicate a significant reduction in SM occupancy due to increased idle compute units caused by GPU communication. As SM occupancy measures the active time of compute units, increased communication inevitably leads to a decrease in the proportion of compute time. In addition, our use of synchronous communication strategy accounts for the latency caused by changes in computing speed between GPUs, further intensifying the decline in SM occupancy. Our results show

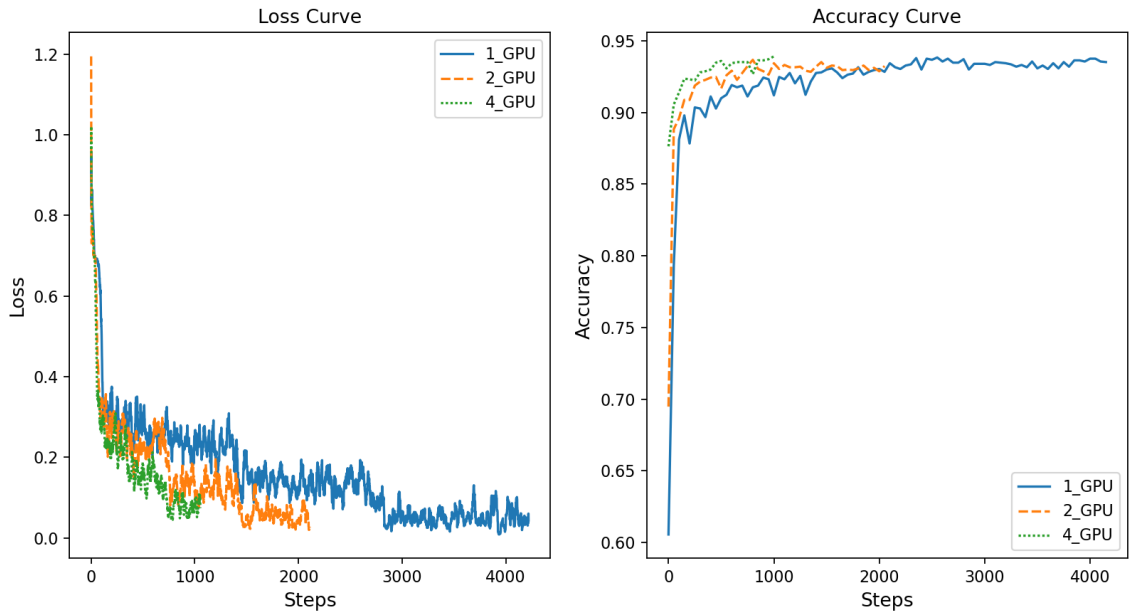

Figure 4.11: Benchmark 6, Data Parallel training loss and accuracy

a 46.07% and 60.39% reduction in SM occupancy when using two and four GPUs, respectively, compared to a single GPU.

We also observe that the accuracy of the distributed trained model's test set has improved. This improvement can be attributed to the principle of model parallelism, where the batch size is maintained the same within one device. With the introduction of more devices, the overall batch size used in the training process increases. A larger batch size enhances the generalization capability of the model and prevents it from falling into the local optimum. Therefore, we consider that the superior performance of the fine-tuned model on 2 and 4 GPUs is due to the larger overall batch size. Figure 4.11 shows the trend of validation loss and accuracy when training the model with different number of GPUs.

4.5.2 Time Consumption in One Training Step

In order to show a more comprehensive distribution of the time taken up by each part of the training process, we conducted tests on 1/2/4 GPUs, and the following Table 4.10 shows the time taken up by each part of the training process.

GPU settings	forward/ms	backward/ms	optimize/ms	step time/ms
1 GPU	11.98	101.85	11.36	137.08
2 GPU	13.05	189.20	11.07	239.78
4 GPU	15.62	229.87	11.44	272.06

Table 4.10: one-step time consumption per GPU

The table above shows that as the number of GPUs increases, there is a slight increase in both forward and backward computation times. However, the increase in

forward time is not significant, while the increase in backward time is more noticeable. Our analysis of the program's execution process suggests that the primary reason for the increase in forward time is the distribution of training data. The distributed training framework performs a Broadcast communication operation in the communication domain, causing this increase. On the other hand, the Optimize process remains consistent at runtime in a multi-GPU context because it requires no communication.

To determine the average time required for gradient Allreduce during backpropagation per encoder, we conducted experiments in two different GPU environments, with 2 and 4 GPUs. Our results show that adding GPUs increases communication time, with an average time of 8.7 ms/encoder and 10.94 ms/encoder for 2 and 4 GPU environments, respectively. The communication time in the 4 GPU environment is about 1.2 times that of the 2 GPU environment, implying a potential communication overhead due to an insufficient number of chunks.

The results suggest that chunking data can help reduce communication overhead, but it also introduces additional overhead. Therefore, it is necessary to find a reasonable number of chunks that minimizes communication overhead while maximizing computational efficiency.

4.5.3 Backpropagation Time Estimation and Measurement

The objective of this section is to estimate the duration of backpropagation in a multi-GPU scenario. We will use the method described in Section 3.4.3 and compare our estimated values with actual measured values.

To achieve this, we need to determine the computation time $D^{compute}$ and the communication time D^{comm} required for each network layer. However, estimating these times accurately is a challenging task. Therefore, we use the actual measured values to validate our estimation approach. Table 4.11 shows the measured values obtained using 2 GPUs.

Layer	$D^{compute}/ms$	D^{comm}/ms
embedding	1.8	22.45
ls_encoder	7.4	8.7
dense	0.513	1.773

Table 4.11: Layer Time Measurement under 2 GPU

The embedding process involves converting each token into a vector and placing it in a position encoder before passing it to the BERT model encoder. The BERT model encoder, which includes a lightseq optimized encoder, is the primary component of the model. Finally, the evaluation layer serves as the final classifier for classification, producing the final prediction results.

Using our proposed estimation method, we estimated the backpropagation time for our model to be approximately 134.853 ms. However, actual measurements yielded

a value of 134.389 ms, resulting in an error of 0.35%. To further validate our method, we performed similar measurements using a 4 GPU configuration. The corresponding data is presented in Table 4.12. Specifically, the calculated backpropagation time for this configuration is 166.27ms, while the actual measured time is 166.233ms, representing an error of 0.02%.

Layer	$D^{compute}/ms$	D^{comm}/ms
embedding	2.22	27.477
ls_encoder	6.94	10.94
dense	0.483	4.698

Table 4.12: Layer Time Measurement under 4 GPU

The results presented above demonstrate that our approach to estimating backpropagation time is relatively accurate. However, it has a limitation in that it requires computation and communication times for each layer of the network, which are often challenging to acquire and prone to substantial errors. As the size of the network increases, this error accumulates, resulting in inaccurate estimates.

4.6 TPU-based training

This section presents the results from training on a TPUv2 using two different frameworks for deep learning, PyTorch XLA and TensorFlow comparing the performance between the two frameworks. As shown in Table 4.13, the baseline performance of the TensorFlow version provides a 40% faster time-to-train compared to the Python XLA version, whilst also maintaining a higher test accuracy. Likewise, the cost is also lower due to the time-to-train being lower. Furthermore, the results from manually enabling bf16 Mixed Precision show that the increase in performance is minimal for the TensorFlow version it still beats out the PyTorch XLA version. The XLA version on the other hand has worse performance when bf16 is enabled but a slightly higher test accuracy.

Framework	Test Accuracy	Time/s	Cost/$	Mixed
PyTorch XLA	92.4%	1207	1.73	FP32
TensorFlow	93.8%	746	1	FP32
PyTorch XLA	93.6%	1443	2	BF16
TensorFlow	92%	677	0.91	BF16

Table 4.13: Benchmark 7, Baseline and Mixed Precision performances for the Python XLA and TensorFlow implementations. Fine-tuning for 3 epochs with a batch size of 16

Another interesting result to look at is how the two different implementations handle memory management. Presented in Table 4.14 is how the batch size impacts

performance, PyTorch XLA does get a marginal performance improvement when increasing to a batch size of 24 but it does run out of memory when increasing the batch size further. Unlike TensorFlow, which allows up upwards of 64 batch size with mixed precision enabled. It's important to note, that the learning rate for the Adamw optimizer was increased as the batch size increased which aided in increasing the final test accuracy.

Framework	Test Accuracy	Time/s	Cost/$	Batch size
PyTorch XLA	93.6%	1443	2	16
PyTorch XLA	93%	1232	1.73	24
PyTorch XLA	-	-	-	32
TensorFlow	92%	677	0.91	16
TensorFlow	93.5%	628	0.83	32
TensorFlow	93.5%	603	0.91	64

Table 4.14: Benchmark 7, Mixed Precision performance with different batch sizes for the two different implementations. Out-of-memory occurred for the XLA implementation at a batch size of 32.

4.7 Overall Evaluation

This section show the results from a systematic evaluation with the performance of all optimization techniques on different devices. Selected devices include NVIDIA V100 16GB, A100 80GB, T4 16GB, A10 24GB and TPUv2 as test platforms. Using a grid search approach and the different optimizations as parameters to measure the time, energy, and cost of training on the selected hardware, all benchmarks are fine-tuned using the pre-trained BERT model on the IMDb dataset for 3 epochs, for parameter details see Table 4.15.

Parameters	Content
Optimizer	AdamW (unfused multi), Adan (fused multi, GPU)
Mixed Precision	FP32, Mixed FP16, Mixed BF16(TPU)
Encoder type	Hugging Face, LightSeq(GPU)
Batch size	GPU(8, 16, 32), TPU(64, 128, 256, 512, 1024)
Hardware	V100, A100, T4, A10, TPUv2, TPUv3

Table 4.15: Parameters for the systematic benchmark.

Figure 4.12 shows four different plots, each focusing on a separate metric. As previously stated, the TPU results are included in graphs not plotting energy-based results due to difficulties of acquiring the energy consumption data.

In plot A, it can be observed that the A100 has the highest training cost but also the lowest energy consumption due to shorter training times. Middle of the pack

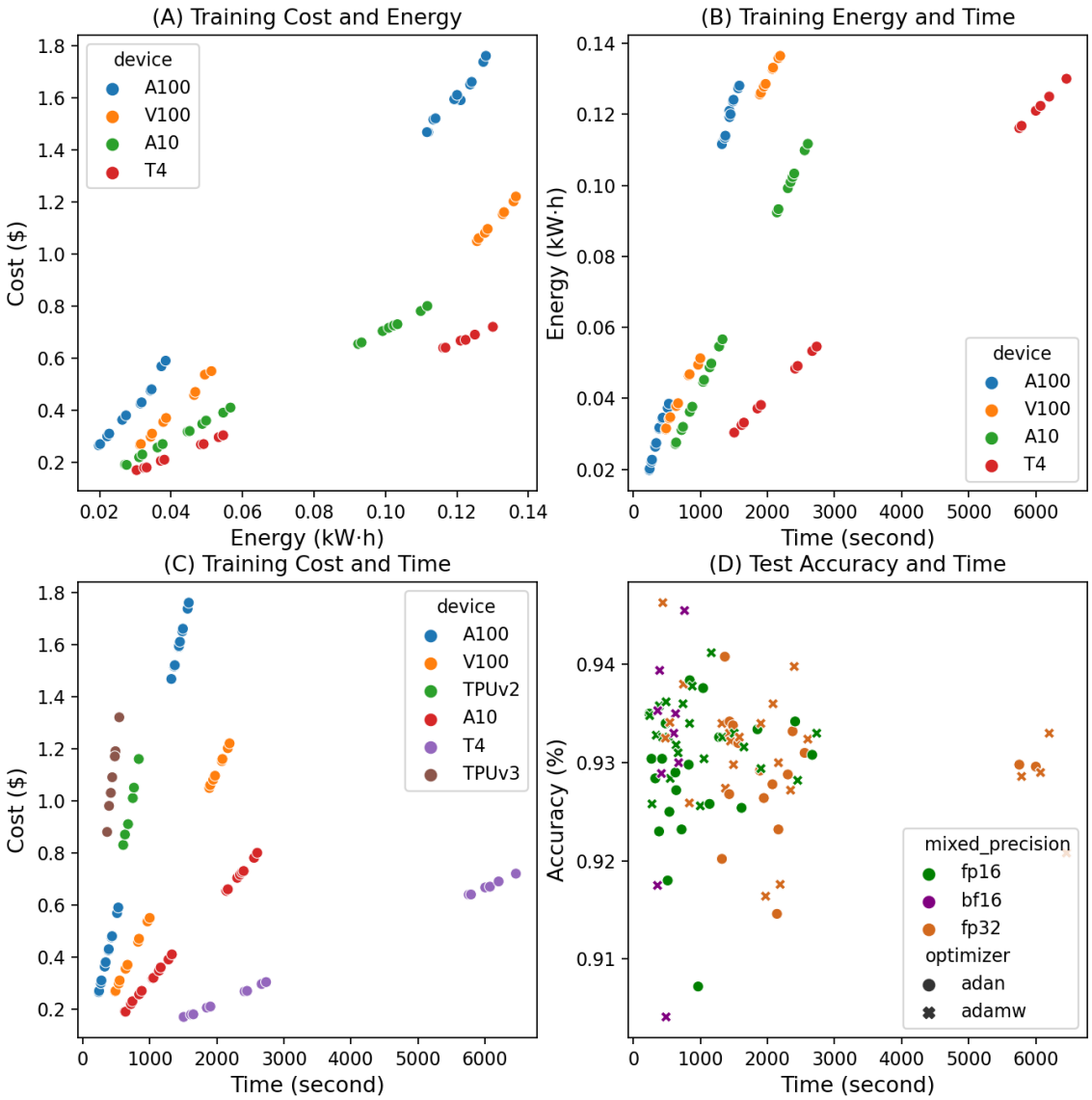

Figure 4.12: Four different plots (A, B, C and D) showing the how the different hardware compares with the optimization techniques previously shown. Each point is one configuration. Due to TPU energy consumption being unavailable in Google Cloud it's only included in plots not plotting energy on any axis.

includes the V100 and A10, with the A10 providing a slightly lower cost as well as lower energy consumption when compared to the V100. Lastly, the T4 has the highest energy consumption due to slow execution but it's the lowest training cost. It's important to note that for a small fine-tuning task the overall differences between the different hardware configurations are less important.

In Figure B, the slope of each data point representing a device segment indicates its average power. It is evident that A100 has an average operating power of 300W, V100 of 250W, A10 of 150W, and T4 of 70W. These differences indicate different application scenarios for different devices. A10 and T4 are designed for deep learning model inference, thus using GDDR6 memory with lower memory bandwidth, where A10 has 600GB/s and T4 has 320GB/s. This allows significant reductions in hardware manufacturing costs while maintaining considerable capacity.

In contrast, A100 and V100 target deep learning model training that requires higher memory bandwidth due to the high communication requirements of the backpropagation process. The memory bandwidth for A100 (40GB) is 1555GB/s A100 (80GB, PCIe) is 1935GB/s, and for V100 (16GB) is 900GB/s. With different types of memory, such as Graphics Double Data Rate (GDDR) and High Bandwidth Memory (HBM), GPUs could achieve the best balance between cost and speed for various deep learning tasks.

Graph C shows the cost relative to training time, although the cost is linear with time it also depends on what the service provider charges. The A100 and TPUv3 are the two fastest configurations in terms of training time, however the TPUv3 is also the most expensive at around twice the cost of an A100. The V100 is the cheapest hardware relative to the training time, with some configurations beating out the TPUv2 in terms of time to train at a substantially lower cost. The most balanced option presented is the A10, with a lower cost than the V100 whilst only being marginally slower to train on. Lastly, the T4 presents the cheapest option but also being a lot slower to train on.

Lastly, graph D highlights the effect of mixed precision on the final test accuracy for every configuration. Showing that the increased performance gain (from mixed precision) that can be seen in the other three graphs has no direct impact on the final accuracy. The graph also shows that the choice of optimizer doesn't impact the final test accuracy in any noticeable manner.

4.7.1 TPU-GPU evaluation

Whilst the previous section highlighted the overall evaluation results, this one focuses on a more direct comparison between the different GPU architectures and TPU architectures. The TPU results shown are based on the Tensorflow implementation whilst the GPU results will be based on both Hugging Face and LightSeq implementations.

The first section in Table 4.16 shows the baseline performances for the different GPUs and TPUs, with the TPUs showing the lowest training time by a large margin. The TPUv3 shows a 50% speed-up over the TPUv2 which is the second fastest in

terms of baseline performance. All of the GPUs are significantly slower when FP32 is enabled, they are however cheaper to utilise with V100 being cheaper than the TPUv3. As well as the A10 and T4 are being cheaper than both of the TPUs but significantly slower. The A10 notably also performs quite closely to the V100 at a much lower cost.

Presented in the second section of Table 4.16 are the mixed precision performances of both the GPUs and TPUs benchmarked. With minor speedups seen for the TPUs, and considerable speedups for the GPUs. With a speedup of 342% in the case of the A100, and between 206% - 259% for V100, A10 and T4. The TPUv3 and A100 are very competitive in terms of training time consumption, however the A100 is half the price. The TPUv2 and V100 show a similiar trend, with nearly the same performance but at nearly half the price. The A10 and the T4 remain the lowest cost options but at lower performance when compared to the other GPUs and TPUs.

Hardware	Test Accuracy	Time/s	Cost/$	Mixed	Module
Google TPUv2	93.8%	746	1.01	FP32	Hugging Face
Google TPUv3	93.25%	478	1.17	FP32	Hugging Face
NVIDIA A100	93.6%	1493	1.66	FP32	Hugging Face
NVIDIA V100	93.9%	2083	1.16	FP32	Hugging Face
NVIDIA A10	93.9%	2399	0.73	FP32	Hugging Face
NVIDIA T4	93.3%	6202	0.69	FP32	Hugging Face
Google TPUv2	92.9%	675	0.91	BF16	Hugging Face
Google TPUv3	92.89%	418	1.03	BF16	Hugging Face
NVIDIA A100	93.4%	436	0.48	FP16	Hugging Face
NVIDIA V100	93.3%	838	0.47	FP16	Hugging Face
NVIDIA A10	94.1%	1162	0.36	FP16	Hugging Face
NVIDIA T4	92.8%	2450	0.27	FP16	Hugging Face
Google TPUv2	94.3%	603	0.83	BF16	Hugging Face
Google TPUv3	93.53%	363	0.88	BF16	Hugging Face
NVIDIA A100	92.5%	277	0.31	FP16	LightSeq
NVIDIA V100	93%	549	0.31	FP16	LightSeq
NVIDIA A10	94.1%	740	0.23	FP16	LightSeq
NVIDIA T4	93.2%	1649	0.18	FP16	LightSeq

Table 4.16: Results based on fine-tuning the BERT model using the IMDb dataset. First section shows the baseline performances for the different architectures, with a batch size of 16 (16x8 for TPUv2/v3). Second section shows the mixed performance of the different hardware with a batch size of 16 (16x8 for TPUv2/v3). Third section features the most optimized implementation for each type of hardware.

The last section in Table 4.16 highlights the best solutions found for the GPUs and TPUs. With the A100 having the lowest training time, 30% faster compared to the TPUv3 at less than half the cost. The V100 similiarly is also 10% faster than the TPUv2 at less than half the cost. Whilst the A10 is slower than both the TPUv2

and V100 it remains competitive due to its very low cost relative to performance. Lastly, the T4 once again remains the by a large margin slowest architecture but at the lowest cost.

5

Discussion

This chapter is dedicated to discussing and highlighting the results, in order to evaluate and form conclusions. The chapter firstly presents a scoring used to evaluate the performance of the different optimization techniques, as well as provide recommendations to end users. Secondly, we test our most optimized solution on a specific test-case, namely a Volvo dataset. Thirdly, we discuss the impact of kernel fusion on time and energy, as well as other software-based techniques for improving the performance. Lastly, we discuss the different hardware used in the thesis, with the last subsection being dedicated to discussing TPUs.

5.1 Comparison between different optimizations

In this thesis, we employed four metrics to evaluate the performance of our methodology. This approach enables us to establish a final criterion for measuring the performance according to the diverse demands of end-users. The computation will be carried out using the formula below:

$$Score = T \times w_1 + C \times w_2 + E \times w_3 - O \times w_4$$

The formula presented uses the variables T for time, C for cost, E for energy, and O for hardware utilization. The Score metric measures the overall performance of the current method, which enables end users to adjust their desired strategy by assigning different weights to individual metrics. A smaller Score value indicates less comprehensive spending and a superior method compared to others. The higher the hardware utilization ratio (occupancy), the better the method's performance.

Considering to diverse user requirements, weights w_1, w_2, w_3, w_4 are assigned varying values to customize the user's ultimate objective. Prior to conducting computations, normalization of all numerical values is undertaken to prevent the final outcomes from being influenced by metric range.

Due to the inability of TPUs to provide comprehensive data in existing workflows and the cost of migrating from Pytorch to TensorFlow, we will not introduce TPUs for comparison in this section. The following Table 5.1 displays the original data obtained from the application of our optimization methods during fine-tuning on the IMDb dataset.

Workload	Time/s	Cost/$	Energy/$kW \cdot h$	Occupancy/%
Unfused Single Adan	2286.85	1.75	0.1425	48.66
Fused Multi Adan	2034.45	1.55	0.1243	46.64
Mixed precision	813.34	0.62	0.0425	58.4
FP16 LightSeq	535.45	0.41	0.0325	41.48
FP16 LightSeq 2GPU	457.55	0.70	0.0389	22.37
FP16 LightSeq 4GPU	281.41	0.86	0.0441	16.43

Table 5.1: Original data from different optimization

We employed the max normalization to rescale all numeric variables to a range of 0 to 1. This normalized data can be utilized for computing scores, the data is presented in Table 5.2.

Workload	Time	Cost	Energy	Occupancy
Unfused Single Adan	1.000	1.000	1.000	0.833
Fused Multi Adan	0.890	0.890	0.873	0.799
Mixed precision	0.356	0.356	0.298	1.000
FP16 LightSeq	0.234	0.234	0.228	0.710
FP16 LightSeq 2GPU	0.200	0.400	0.273	0.383
FP16 LightSeq 4GPU	0.123	0.492	0.309	0.281

Table 5.2: Normlized data from different optimization

There are four indicators in each strategies: time, cost, energy, and hardware occupancy. Indicators that need to be ranked are time, cost, and energy consumption. The hardware occupancy uses a fixed weight and therefore does not need to participate in the ranking process. There are four weights in total, w1, w2, w3, and w4, with w1 being the weight of time, w2 being the weight of cost, w3 being the weight of energy consumption, and w4 being the weight of hardware occupancy, which is fixed at 0.1. In the balance strategy, w1, w2, and w3 are all 0.3. Apart from balance, each strategy corresponds to three weights in descending order, namely 0.5, 0.3, and 0.1. The three indicators for each strategy correspond to the relative weight of the current indicator.

	w1	w2	w3	w4
balance	0.3	0.3	0.3	0.1
time_cost_energy	0.5	0.3	0.1	0.1
time_energy_cost	0.5	0.1	0.3	0.1
cost_time_energy	0.3	0.5	0.1	0.1
cost_energy_time	0.1	0.5	0.3	0.1
energy_time_cost	0.3	0.1	0.5	0.1
energy_cost_time	0.1	0.3	0.5	0.1

Table 5.3: Weight Table

Based on the data in Table 5.3, we calculated the score for each strategy using the score formula. A lower score indicates better performance for the current configuration under the respective strategy.

	Unfused Adan	Fused Adan	FP16	FP16_ls	2GPU	4GPU
balance	0.817	0.716	0.203	**0.138**	0.224	0.249
time_cost_energy	0.817	0.719	0.214	**0.139**	0.209	0.212
time_energy_cost	0.817	0.716	0.203	**0.138**	0.184	0.175
cost_time_energy	0.817	0.719	0.214	**0.139**	0.249	0.286
cost_energy_time	0.817	0.716	0.203	**0.138**	0.264	0.323
energy_time_cost	0.817	0.712	0.191	**0.137**	0.198	0.213
energy_cost_time	0.817	0.712	0.191	**0.137**	0.213	0.286

Table 5.4: Score Table, the lower the better

5.1.1 Recommendation

When not considering using dedicated accelerators, using LightSeq, mixed precision, and fused Adan achieved the best overall performance across all strategies within the given weight. These results demonstrate the effectiveness of the optimization techniques we used in different strategies. Therefore, in general, we recommend all users to optimize their models using LightSeq, enable mixed precision training, and try using fused kernels for their optimizer (the more commonly used CUDA fused kernels for Adam and AdamW will be supported in PyTorch 1.13 and later version).

5.2 Volvo Dataset

The dataset provided by the Volvo Group's Service Market Logistics consists of three attributes, namely Part ID, Part Description, and Tariff code. The objective of this task is using the Part Description attribute predict the corresponding tariff code.

A tariff code is used for used for customs compliance in Harmonized System [38] for the export process for goods. In the Volvo dataset, full prediction of tariff codes

is not performed. Instead, the corresponding tariff code for a particular level is predicted based on the directory hierarchy. The term "level" in this context refers to the number of levels in the directory, where higher levels correspond to more detailed directories and more detailed tariff codes. In our thesis, predictions were made using tariff level of 10, which comprised 475 different classes.

5.2.1 Benchmark results

In this section, we compare the performance of training configurations between the workflow used in Volvo and our optimized single GPU training configuration. Volvo using the unfused AdamW optimizer and Huggingface pre-trained BERT model to train on the Volvo dataset using FP32 for 3 epochs. We will employ the LightSeq optimized BERT pre-trained model and Fused Adan optimizer to train on the Volvo dataset with mixed precision enabled for 3 epochs. We compare differences in training time, energy consumption, and cost, and also focus on the accuracy performance of the final model on the test set.

In order to mitigate the impact of validation on training time and energy consumption, we perform validation at the end of each epoch. This practice ensures that the majority of energy consumption is attributed to the training process while maintaining the accuracy of training time by subtracting the time spent on validation from the total training time. Such approach reduces the validation overhead and helps in managing the energy consumption during training epochs. Therefore, the effectiveness of the training process is maximized while minimizing the overall computational costs.

On the Volvo dataset, a noticeable decrease in test accuracy was observed by the model optimized through Lightseq, using the same hyperparameters as used on the IMDb dataset. The main reason for the observed decrease in test accuracy in Lightseq is believed to be the high sensitivity of the optimized model to different learning rates on the Volvo dataset. This phenomenon will be discussed in the next subsection.

In order to achieve a more stable initial training for the model and mitigate the negative effects of excessively high learning rates in the early stages of the training process, we decided to increase the number of warmup steps from 50 to 2000. Further, we also reduce the global learning rate from $1 \cdot 10^{-4}$ to $7 \cdot 10^{-5}$. Consequently, the optimized LightSeq model demonstrated an improvement in accuracy of 0.79, which is comparable to the baseline. The tuning strategy of hyperparameters will be demonstrated in following subsection.

Workload	Time/s	Cost/$	Energy/mJ	Occupancy/%	Test accuracy
Optimized	657.79	0.50	133333501	41.47	0.79
Original	2507.27	1.91	518474890	48.03	0.81

Table 5.5: Performance on Volvo dataset

It can be observed that when using the model optimized with LightSeq, along with the combined usage of fused Adam and mixed precision optimization, results in a reduction of training time by 73.76%, a decrease in energy consumption by 74.28%. Although the test accuracy decreased by 2%, we think it is acceptable and it is possible improving the test accuracy by tuning the hyperparameters of the Adan optimizer which are not carefully tuned currently.

5.2.2 Switch to New Datasets

This section focuses on investigating the failure of a trained model on the Volvo dataset after migrating hyperparameters used on the IMDb dataset. The observed decline in the accuracy of the model on the test set after three training epochs prompted a investigation of various factors that may lead this problem. We test LightSeq, mixed precision, the number of classifications in the dataset, and hyperparameters of the optimizer to identify the root cause of the problem.

The possible reasons for the nan loss were identified as the presence of nan data in the dataset and the setting of the learning rate too large. However, the removal of nan data in the preprocessing of the dataset was already done before we start training, and training using the Hugging Face encoder in the same scenario is successful. Drawing a conclusion, we believe that this problem might be issued with learning rate.

We investigated possible solutions to the identified problem. Two methods were tested, namely adjusting the learning rate from $1 \cdot 10^{-4}$ to $7 \cdot 10^{-5}$ with the addition of warm-up, and the other one is tuning optimizer's β_3. The former method involved the use of a smaller learning rate at the beginning of the model fine-tuning, followed by a larger learning rate to enhance the learning speed after the model parameters stabilized. The latter method was designed to enhance the adaptability of the optimizer to the learning rate, given that the initial value of $\beta_3 = 0.9$ (used in initial test on Volvo dataset) may hinder the rapid adjustment of the learning rate during optimization. Changing β_3 to $\beta_3 = 0.99$ in the adjusted optimizer helped solve this problem. Both methods proved effective in addressing the problem of model training failure, resulting in a final test set accuracy of 0.79.

5.3 Kernel Fusion

Section 4.2 highlight the effectiveness of Kernel Fusion in improving kernel performance and reducing energy consumption. This method eliminates repetitive and unnecessary operations, thus optimizing the data transfer process. Research [39] has shown that data movement can have a significant impact on the time and energy consumed in computer systems.

Kernel fusion is a technique that combines multiple code blocks that generate overhead within one block. As a result, kernel resource allocation and data transfer overhead are minimized, reducing associated instructions, time, and energy consumption.

Despite the benefits of kernel fusion, the technique has drawbacks. For example, encapsulating multiple operations in a complex kernel can reduce flexibility, which may pose challenges during model development. Meanwhile, if the kernel has defects and requires analysis, complicated code blocks could be a problem with the debugging process. To enhance the readability and maintainability of the program, it is important to follow software engineering principles while implementing kernel fusion.

In conclusion, the appropriate timing of kernel fusion is critical. Establishing a stable model structure before developing a high-performance kernel is essential to optimize performance and energy efficiency while minimizing subsequent maintenance costs. This approach is ideal because significant model changes are unlikely to occur, and modifications to fused kernels might be costly.

5.4 Software and Hardware Improvements for ML

In this section we discuss different approaches for optimizing machine learning training performance, describing the phenomena we observed in overall evaluation at the software as well as hardware level.

5.4.1 Software Improvements

Fused Adan shows potential improvement in hardware utilization. It could significantly improve the kernel active ratio from 18.84% to 99.81% for unfused single tensor access and fused multi tensor access respectively. This technique can help reduce idle time caused by kernel launch overhead.

Furthermore, Figure 5.1 shows that with the use of mixed precision training and LightSeq Encoder on V100, our batch size can be increased to 32, which cannot be achieved when using Hugging Face encoder. This demonstrates the improvement in memory usage brought by software optimization. Compared to using A100, Hugging Face encoder, and a batch size of 16, the training time and energy consumption gap between V100 and A100 were reduced by LightSeq. We can also observe the significant enhancement in performance achieved by LightSeq on the V100. When compared with the HuggingFace Encoder, the LightSeq Encoder can reduce the training time by a substantial 71.37%.

These results are interesting for another reason as well, they show a kind of "generational uplift", in which the older generation (V100) can be competitive with newer generation cards (A100) through CUDA-based optimizations. Naturally, given access to both an A100 and V100 one should select the A100 however, it shows that if V100 is all that's accessible one can still expect to be able to squeeze out performance in the ballpark of a unoptimized A100.

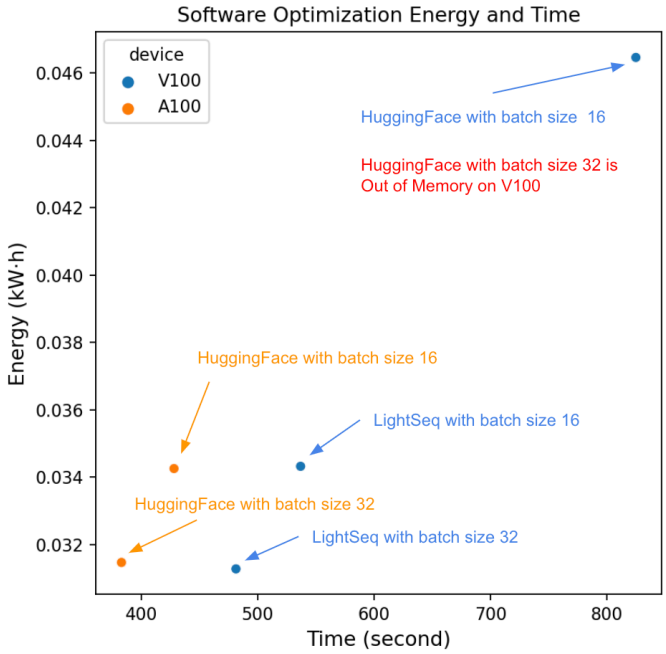

Figure 5.1: LightSeq, HuggingFace on V100 and HuggingFace on A100 with batch size of 16, 32

5.4.2 Different Hardware

We conducted a benchmark test on various devices, and we will discuss the results based on the cost and training time. Each point on Figure 5.2 represents the optimal training time and cost achieved on the respective device.

Our findings indicate that training costs for both A100 and V100 are roughly the same, at about $0.27 per unit. However, the A100 outperformed the V100 in training speed, taking only 238.52 seconds compared to the latter's 481.41 seconds. The training time for TPUv2 is comparable to that of A10, taking about 600 seconds. Similarly, the TPUv3's training efficiency is between A100 and V100, with a training time of 360 seconds. However, the use of TPUs incurs high costs, ranging from about $0.8 to $0.9. In contrast, T4 offers the most affordable training costs, with a minimum cost of $0.17, despite having the longest training time of 1504.33 seconds. Since T4 is primarily designed for model inference, the training process is constrained by memory bandwidth, resulting in suboptimal results.

5.4.3 Mixed Precision on GPUs

Furthermore, we observed significant improvements in time and energy consumption through the use of mixed precision training on GPUs, which was made possible by fully leveraging the tensor core in NVIDIA GPUs. This specialized core is capable of accelerating the computation of FP32-FP16 mixed precision training.

Figure 5.3 shows a significant reduction in training time and energy consumption when mixed precision training is enabled. This is mainly due to the Tensor cores

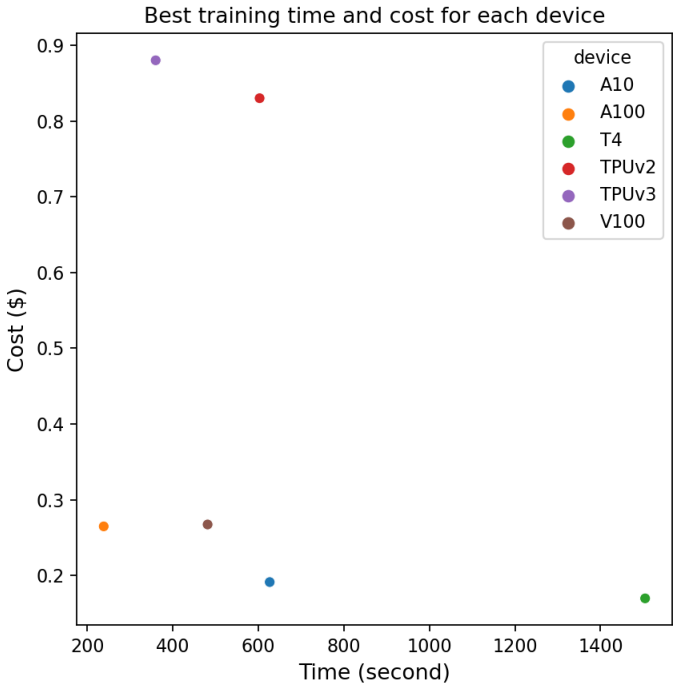

Figure 5.2: Best training time and the cost on different devices.Training configuration: LightSeq Encoder(GPUs), Fused Adan(GPUs) and AdamW(TPU), Mixed Precision Training. Batch size 32 using on A10, A100, V100. Batch size 16 using on T4. Batch size 512 using on TPUv2 and TPUv3.

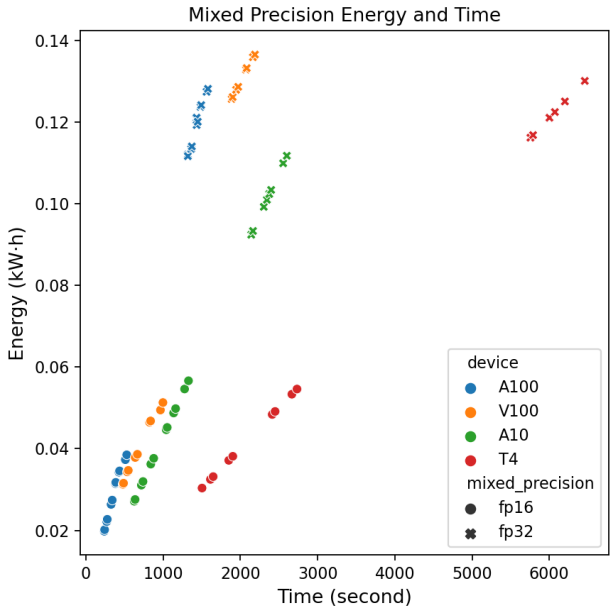

Figure 5.3: Different training time and energy when enable and disable mixed precision on GPUs

present in NVIDIA GPUs. While FP32 training uses streaming multiprocessors for computation, mixed precision training uses Tensor cores, resulting in improved performance and energy efficiency.

Similar to tensor cores, specialized acceleration units can provide various computing power for increasingly diverse AI models, and improvements in dedicated acceleration units have not stopped. NVIDIA has introduced the Transformer Engine in its latest H100 chip to speed up large language models, which are widely used. We can see a trend where hardware vendors are increasingly adding specialized computing units to general-purpose computing chips or designing computing chips dedicated to specific AI workloads to achieve better computing performance or energy efficiency.

We expect to see more specialized computing chips bring more possibilities to the field of machine learning in the future or the emergence of a specialized computing model dedicated to the field of machine learning, which maximizes computing performance while maintaining generality.

5.5 Randomness in Test Accuracy

During benchmarking, it was observed that running the fine-tuning process with the same parameters on the same device resulted in slightly different test accuracy. Theoretically, fixing the random seed should eliminate this problem, and the potential reasons behind this phenomenon need to be investigated. In the benchmark, custom kernels were employed in the fused Adan and lightseq encoder, which are not affected by Pytorch's random seed. The fused Adan does not use curand, the random number interface provided by NVIDIA CUDA. Therefore, the differences between the lightseq encoder and hugging face encoder were compared, with other parameters held constant.

After running lightseq and hugging face five times each, it was observed that hugging face results were consistently the same, while lightseq showed inconsistencies in test accuracy. We observed that the accuracy of the model fluctuated between 0.9226 and 0.9386 when using Lightseq Encoder, while the accuracy of the model using Hugging Face was consistently 0.9298.

The kernels used by hugging Face were implemented using Pytorch, so the random seed could be fixed using Pytorch's interface. However, upon checking the documentation for Lightseq, no description of the random seed was found. Further inspection of the Lightseq kernel's cuda file revealed that curand was called inside the kernel. No seed setting parameters were found in the kernel's Python interface. Therefore, it can be concluded that the randomness observed in lightseq from curand in its custom kernel.

5.6 TPU

This section is for discussing specifics regarding TPU insights and problems. Such as the performance differences between the two training frameworks, or between

TPUs-GPUs.

5.6.1 Framework

The aim was to seamlessly migrate the training process from GPUs to TPUs, which was first done using the Accelerate Library by Hugging Face which supports TPU as the accelerator. However, there was a problem encountered when attempting to utilise the BF16 mixed precision mode in the Accelerate library which led to NaN loss. An investigation was conducted in order to ensure that the NaN loss wasn't caused by the selected model, dataset or training workflow.

Firstly, the Accelerate library which was investigated by using one of their official examples and enabling BF16 mixed precision. But the same NaN loss problem remained which most likely entails that the error is in the Accelerate library. Secondly, by enabling BF16 mixed precision manually in XLA rather than through Accelerate. However, there was no difference in performance between the baseline and the mixed precision workflow.

The second approach was using TensorFlow, which as mentioned before is different from PyTorch both in workflow but also performance. TensorFlow, like the PyTorch implementation uses a pre-trained BERT model from the transformers library by Hugging Face, as well as the IMDb dataset provided by the datasets library. Enabling BF16 mixed precision is done by changing the mixed precision policy.

The problems in using PyTorch XLA could be seen in Table 4.13 in the Results Chapter, where the PyTorch XLA surprisingly has a worse performance with BF16 enabled.

5.6.2 TPU-GPU

The results show that when using mixed precision on the GPUs they become competitive with the TPUs, both the A100 and V100 catch up to the TPUv2 and TPUv3 respectively whilst remaining roughly 50% cheaper to use. These results are further exacerbated when comparing the TPU performances with the optimized GPU performances, where we can see a significantly faster training time for the A100, even compared to the v3. Its important to note however that not many optimization steps were carried out for the training on TPUs due to the thesis primarily focusing on CUDA-based optimizations. The main optimizations performed on the TPUs were to utilize mixed precision (bf16) in addition to maximizing the batch size, although TPUs naturally make use of kernel fusion though the XLA compiler (included in TensorFlow by default).

As shown in Table 4.16, the TPUv2 and TPUv3 perform well relative to the different GPU architectures benchmarked. As previously stated in Chapter 3 the main competitors are the TPUv3 and A100 as well as TPUv2 and V100 but due to the surprising performance of the A10 it may also be comparable to both the TPUv2 and V100. As for the TPUv3 and A100, we can see that when only utilizing mixed precision the two are pretty much the same in terms of time, but the A100 is half

the cost of a TPUv3 making it a more efficient option in general. The TPUv2 outperforms both the V100 and A10 by a rather large margin when only comparing mixed precision training time, although at nearly 2-3x the price.

The major shift occurs when replacing the original Hugging Face BERT implementation with the LightSeq BERT implementation on the GPUs. Showcasing the performance increase that can be gained through CUDA-optimized implementations, with the V100 outperforming the TPUv2 in terms of both training time and cost, and even the A10 being 25% slower at 28% of the cost. A similar trend holds for the A100 and the TPUv3, the A100 performs 30% faster than the TPUv3 whilst only being 30% of the cost.

Even though the TPUs can handle way larger batch sizes (upwards of 8*128 on the TPUv3), which helps to decrease the training time significantly; they still cant compete with the GPUs in terms of both cost and time. However, as previously stated, the TPUs barely had any optimizations performed by us compared to the GPUs, so there could be architectural changes that may increase their performance significantly. Similarly to LightSeq for CUDA-based GPUs which showed speedups of upwards of 50% in some cases.

Another thing to consider, that whilst the TPUv3 is framed as a competitor to the A100, theres still roughly 2 years between their releases, thus the TPUv4 might be a better competitor seeing as it was released in 2021. But due to the cost of using Google Cloud services, it wasnt considered for benchmarking in this thesis.

5.6.3 Unfair Batch Sizes

When comparing the GPUs and TPUs, different batch sizes were experimented on, in particular 8, 16, 32 for GPUs and 8x(8, 16, 32, 64, 128) for TPUs. The GPU batch sizes were determined by progressively scaling up the batch size on a NVIDIA V100, as it served as the baseline GPU. The batch size for TPUs were determined in the same manner, namely progressively scaling up the batch size on a TPUv2. However, this leads to the following question: Is it a fair comparison between the GPUs and TPUs? Seeing as the A100, and most likely the A10 both could've utilised 64 or higher batch size but it was never tested. Even with limitations in place the A100 showed the best performance of all hardware benchmarked, with 64 or 128 batch size it's likely that the time-to-train on the A100 could've decreased by roughly 35-40%. Whilst impressive, the actual result isn't as relevant seeing as the A100 was already the best performing hardware.

What's more important is to consider the A10 with the potential for 64 batch size, seeing as it was already close in terms of performance with both the TPUv2 and V100, at a much lower cost. The main reason for not examining the A10 with 64 batch size comes down to time constraints. As the overall evaluation was started late into the thesis work (was not originally planned, but the original plan had to be re-worked due to implications).

Overall, regarding the A100, results with 64 or 128 batch size would've been interesting to see although the actual results (i.e it being the best of the surveyed hardware)

wouldn't have changed. The results of 64 batch size on the A10 would've been both very interesting and important. Seeing as the A10 was actually only marginally slower than the TPUv2 with 32 batch size, and at a fraction of the cost it most likely would've outperformed the TPUv2 if tested. Likewise, it would've most likely outperformed the V100 in terms of price relative to time, and potentially been close in time.

It's of course important to remember that this hypothetical, since it wasn't tested. It's also highly probable that the A10 would've ran out of memory, seeing as the V100 with 60% of the memory ran out of memory at right after 32 batch size, and going from 32->64 batch size doubles the required memory.

6

Future work

This chapter provides insights into future work based on the limitations and/or results of this thesis.

6.1 Gradient Compression

In distributed training, we have observed significant communication delays caused by gradient communication during the backpropagation process, which greatly reduces hardware utilization and system throughput. Therefore, we can attempt to modify the algorithm from the perspective of reducing communication time to reduce the communication volume of distributed training. Lin et al. [40] proposed a gradient compression theory, which suggests that 99% of gradient information is redundant during model training. Thus, gradient information can be compressed to reduce the amount of data propagated by gradients in the system. Based on the above theory, Tang et al. [41] developed an algorithm called 1-bit Adam, which involves running a warm-up process and setting v_t to a fixed value after stabilizing the optimizer. At this point, Adam algorithm degenerates into SGD, and gradient error correction can be used to prevent gradient descent errors. We also expect to see similar phenomena in the Adan optimizer optimization process. Therefore, developing 1-bit Adan could be a feasible way to reduce system communication in distributed training while achieving faster convergence speed.

6.2 $D^{compute}$ and D^{comm} Estimation

In Section 3.4.2, we propose a method for estimating backpropagation time. This algorithm has two key parameters, $D^{compute}$ and D^{comm}. However, the current use of actual measurements of these parameters limits our knowledge of their actual values during the theoretical analysis phase. In fact, it is possible to estimate $D^{compute}$ and D^{comm} based on the model's learnable parameter quantity and structure.

OpenAI [42] proposes a method for estimating model computing requirements by providing calculations for commonly used models. Marius et al. [43] suggests that the ratio of computational requirements between forward and backward propagation is approximately 2:1. Therefore, based on the computational requirements of forward propagation and taking into account the theoretical peak computing capability of our hardware, we can roughly estimate the value of $D^{compute}$ and adjust it based

on the actual computing capability when in use. Furthermore, when the number of trainable parameters in the model is determined, the communication requirements of D^{comm} are also established. We can estimate the value of D^{comm} by factoring in the number of communication nodes and data chunking in the system.

Approximately 110 million trainable parameters are contained in the BERT base, each of which generates a gradient that needs to be acquired by all GPUs. The total amount of data required for transmission in this scenario is approximately 110M*sizeof(float). If the gradient is stored in FP16, the sizeof(float) is 2 bytes, and if saved in FP32, the sizeof(float) is 4 bytes. Taking FP16 as an example, the total data is estimated to be about 220 megabytes. Based on 2(N-1)*K/N, which is the total data that needs to be transmitted in Ring All reduce, N is the number of GPUs and K is the size of the data. The amount of data to be transmitted is expected to approach 2K as the number of GPUs increases. This equates to about 440 megabytes. The theoretical bandwidth of V100 PCIe is 900 gigabytes per second, which means that transmitting 440 megabytes of data takes about 5ms. The actual transmission speed may, of course, be affected by many factors.

6.3 Performance Improvement on Fused Adan

In the Fused Adan kernel micro benchmark, we observed a performance drop problem in Kernels A and C due to memory access bottlenecks. One possible solution is to pre-fetch data using shared memory and a vector instruction to retrieve data required by adjacent threads and store it in shared memory. This can reduce the number of LDG instructions in the SM unit and improve system throughput. However, vector instructions introduce additional overhead, and we need to evaluate whether this overhead or the memory access bottleneck has a greater impact on performance.

6.4 System Level Energy Measurement

In this thesis, we only measured GPU energy consumption. However, it is important to note that, in practice, servers used for machine learning require substantial power to dissipate, as the risk of performance degradation due to device overheating increases. Data center GPUs typically use passive cooling to regulate temperature, making information such as fan speed and power consumption unavailable through NVIDIA's software libraries.

To minimize energy costs associated with heat dissipation and CO2 emissions, Microsoft has explored deploying its servers in seawater, which is called Project Natick [44]. This involves circulating cooler seawater through server pods to remove heat, resulting in a significant reduction in energy consumption associated with dissipating heat from servers.

Analyzing energy consumption at the system level can provide insights into the potential benefits of large-scale applications such as Project Natick. This knowledge

can assist data centers focused on machine learning in their efforts to achieve green computing and reduce carbon emissions.

6.5 TPU energy measurements

In this thesis we aimed to provide energy measurements, as an area of improvement. However, Google doesn't offer any tools for measuring the energy consumption for TPUs directly (unlike NVIDIA's SMI). Thus, the energy measurements conducted in the thesis remain inconclusive for the TPU sections, which makes it more difficult to compare them with GPUs. Although we did find a paper by Google on the TPU archteciture which details energy consumption relative to a NVIDIA V100 [29], it's not apt to use extrapolated values in our final results and recommendations, which is why we did not utilise them.

Future work within the field should focus on developing methods and tools for accuractly evaluating the energy consumption of TPUs, Like NVIDIA whom provides tools for accurately measuring the energy consumption for their GPUs since a few generations back. Developing tools for measuring the energy consumption of TPUs can be done either by researchers or directly by Google. Such tools could help improve the sustainability of cloud computing, by allowing developers to optimise software to make it more energy efficient when using TPUs.

7

Conclusion

This thesis examines various hardware-related methods for accelerating fine-tuning machine learning models, measuring their impact on time, energy consumption, and cost. We analyze how these methods differ in hardware utilization and fine-tune the pre-trained BERT-base model on the IMDb dataset using kernel fusion, multi tensor access, mixed precision, and distributed training techniques. We also introduce a fine tuning scheme using Google TPU and adapt our original workflow to run seamlessly in a TPU environment with TensorFlow and PyTorch XLA. Our evaluation compares the optimization of a single GPU on V100 and A100 with the fine tuning workflow for a single TPU. We discuss which optimizations are best suited for different user scenarios. Our results show that fine tuning using LightSeq, Fused Adan, and mixed precision achieves optimal performance in all scenarios. We demonstrate how the optimized workflow reduces time, energy, and cost compared to the existing workflow for the Volvo dataset using optimal training settings.

We optimized the Adan optimizer kernel by implementing kernel fusion and multi-tensor access. These techniques effectively minimized overhead and raised the Adan kernel's active ratio compared to its vanilla implementation, from 18.48% to 99.81%. Notably, we observed a significant improvement in mixed precision training, with a 60% reduction in time and 66% savings in energy consumption. This suggests that using specialized machine learning processing units can lead to improved system performance and potential higher energy efficiency. Although we observed gains from distributed data parallelism in distributed training, the training speedup ratio did not increase linearly with the number of GPUs when fine-tuning the pre-trained BERT model. Our backpropagation time estimation showed that gradient communication in backpropagation had the most significant impact on running time, highlighting the need for optimization in gradient communication between devices. While we achieved similar performance in TPU fine tuning as in GPU training configuration on training time, while the training energy remains unknown. We also observe widely varying TPU training performance across TensorFlow and Pytorch XLA platforms, imply that not stable software may cause unexpected training performance.

Concluding with the results obtained, the following research questions could be answered:

1. How can the fine-tuning process of the BERT model benefit from hardware-aware optimization techniques in terms of time, energy consumption and mon-

etary cost?

2. What are the benefits and drawbacks of using GPU-based training methods as opposed to a TPU-based training method?

For the first problem, we optimized our workflow using LightSeq, Fused Adam, and mixed precision for fine-tuning the pre-trained BERT model. This configuration outperformed the one using HuggingFace, unfused AdamW, and mixed precision disabled. When applied to the Volvo dataset, our optimized configuration resulted in 73.76% time and cost savings while reducing energy consumption by 74.28%.

Regarding the second problem, we found that TPU-based training did not offer a significant advantage over GPU-based training for fine tuning time. In addition, we cannot compare the energy consumption of TPU and GPU training because we do not have access to TPU's energy consumption from the software side. Considering the cost of migrating to a TPU workflow and learning costs, we concluded that TPU settings do not provide significant benefits in our fine-tuning task. Therefore, we still recommend using GPUs for pre-trained model fine tuning.

Bibliography

[1] OpenAI, *Introducing chatgpt*, `https://openai.com/blog/chatgpt`.

[2] A. Ramesh, P. Dhariwal, A. Nichol, C. Chu, and M. Chen, "Hierarchical text-conditional image generation with clip latents," *arXiv preprint arXiv:2204.06125*, 2022.

[3] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[4] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[5] P. Nayak, *Understanding searches better than ever before*, `https://blog.google/products/search/search-language-understanding-bert/`.

[6] K. Cho, B. Van Merriënboer, C. Gulcehre, *et al.*, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[7] G. Lauterbach, "The path to successful wafer-scale integration: The cerebras story," *IEEE Micro*, vol. 41, no. 6, pp. 52–57, 2021.

[8] A. Paszke, S. Gross, F. Massa, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[9] M. Abadi, "Tensorflow: Learning functions at scale," in *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, 2016, pp. 1–1.

[10] G. Tan, L. Li, S. Triechle, E. Phillips, Y. Bao, and N. Sun, "Fast implementation of dgemm on fermi gpu," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, pp. 1–11.

[11] S. Mittal and S. Vaishay, "A survey of techniques for optimizing deep learning on gpus," *Journal of Systems Architecture*, vol. 99, p. 101 635, 2019.

[12] P. Micikevicius, S. Narang, J. Alben, *et al.*, "Mixed precision training," *arXiv preprint arXiv:1710.03740*, 2017.

[13] S. Shi, Q. Wang, P. Xu, and X. Chu, "Benchmarking state-of-the-art deep learning software tools," in *2016 7th International Conference on Cloud Computing and Big Data (CCBD)*, IEEE, 2016, pp. 99–104.

[14] V. Sze, Y.-H. Chen, J. Emer, A. Suleiman, and Z. Zhang, "Hardware for machine learning: Challenges and opportunities," in *2017 IEEE Custom Integrated Circuits Conference (CICC)*, IEEE, 2017, pp. 1–8.

[15] Y. Wang *et al.*, "Performance and power evaluation of ai accelerators for training deep learning models," *arXiv preprint arXiv:1909.06842*, 2019.
[16] S. Ding, J. Shang, S. Wang, *et al.*, "Ernie-doc: A retrospective long-document modeling transformer," *arXiv preprint arXiv:2012.15688*, 2020.
[17] M. Marcus, B. Santorini, and M. A. Marcinkiewicz, "Building a large annotated corpus of english: The penn treebank," 1993.
[18] T. Brown, B. Mann, N. Ryder, *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.
[19] J. L. Elman, "Finding structure in time," *Cognitive science*, vol. 14, no. 2, pp. 179–211, 1990.
[20] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.
[21] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.
[22] K. You, M. Long, J. Wang, and M. I. Jordan, "How does learning rate decay help modern neural networks?" *arXiv preprint arXiv:1908.01878*, 2019.
[23] H. Robbins and S. Monro, "A stochastic approximation method," *The annals of mathematical statistics*, pp. 400–407, 1951.
[24] C. Darken and J. Moody, "Note on learning rate schedules for stochastic optimization," *Advances in neural information processing systems*, vol. 3, 1990.
[25] X. Xie, P. Zhou, H. Li, Z. Lin, and S. Yan, "Adan: Adaptive nesterov momentum algorithm for faster optimizing deep models," *arXiv preprint arXiv:2208.06677*, 2022.
[26] A. Botev, G. Lever, and D. Barber, "Nesterov's accelerated gradient and momentum as approximations to regularised update descent," in *2017 International joint conference on neural networks (IJCNN)*, IEEE, 2017, pp. 1899–1903.
[27] N. P. Jouppi, C. Young, N. Patil, *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th annual international symposium on computer architecture*, 2017, pp. 1–12.
[28] H.-T. Kung, "Why systolic architectures?" *Computer*, vol. 15, no. 1, pp. 37–46, 1982.
[29] N. P. Jouppi, D. H. Yoon, G. Kurian, *et al.*, "A domain-specific supercomputer for training deep neural networks," *Communications of the ACM*, vol. 63, no. 7, pp. 67–78, 2020.
[30] X. Wang, Y. Xiong, Y. Wei, M. Wang, and L. Li, "Lightseq: A high performance inference library for transformers," *arXiv preprint arXiv:2010.13887*, 2020.
[31] X. Wang, Y. Wei, Y. Xiong, *et al.*, "Lightseq2: Accelerated training for transformer-based models on gpus," *arXiv preprint arXiv:2110.05722*, 2021.
[32] M. Li, D. G. Andersen, J. W. Park, *et al.*, "Scaling distributed machine learning with the parameter server," in *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, 2014, pp. 583–598.
[33] A. Sergeev and M. Del Balso, "Horovod: Fast and easy distributed deep learning in tensorflow," *arXiv preprint arXiv:1802.05799*, 2018.

[34] B. Recht, C. Re, S. Wright, and F. Niu, "Hogwild!: A lock-free approach to parallelizing stochastic gradient descent," *Advances in neural information processing systems*, vol. 24, 2011.

[35] G. Wang, Y. Lin, and W. Yi, "Kernel fusion: An effective method for better power efficiency on multithreaded gpu," in *2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*, IEEE, 2010, pp. 344–350.

[36] A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. Potts, "Learning word vectors for sentiment analysis," in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, Portland, Oregon, USA: Association for Computational Linguistics, Jun. 2011, pp. 142–150. [Online]. Available: http://www.aclweb.org/anthology/P11-1015.

[37] P. Mattson, C. Cheng, C. Coleman, *et al.*, *Mlperf training benchmark*, 2019. arXiv: 1910.01500 [cs.LG].

[38] *Harmonized system (hs) codes*, https://www.trade.gov/harmonized-system-hs-codes.

[39] G. Kestor, R. Gioiosa, D. J. Kerbyson, and A. Hoisie, "Quantifying the energy cost of data movement in scientific applications," in *2013 IEEE international symposium on workload characterization (IISWC)*, IEEE, 2013, pp. 56–65.

[40] Y. Lin, S. Han, H. Mao, Y. Wang, and W. J. Dally, "Deep gradient compression: Reducing the communication bandwidth for distributed training," *arXiv preprint arXiv:1712.01887*, 2017.

[41] H. Tang, S. Gan, A. A. Awan, *et al.*, "1-bit adam: Communication efficient large-scale training with adams convergence speed," in *International Conference on Machine Learning*, PMLR, 2021, pp. 10 118–10 129.

[42] OpenAI, *Ai and compute*, https://openai.com/research/ai-and-compute.

[43] J. Marius, *Whats the backward-forward flop ratio for neural networks?* https://www.lesswrong.com/posts/fnjKpBoWJXcSDwhZk/what-s-the-backward-forward-flop-ratio-for-neural-networks.

[44] Microsoft, *Microsoft finds underwater datacenters are reliable, practical and use energy sustainably*, https://news.microsoft.com/source/features/sustainability/project-natick-underwater-datacenter/.

A

Appendix 1

A.1 Fused Adan

To access Fused Adan kernel, please refer to following Github repository:
Adan: Adaptive Nesterov Momentum Algorithm for Faster Optimizing Deep Models

A.2 Benchmark Framework

To access our benchmark framework, please refer to following Github Repository:
Master Thesis Benchmark Framework

A.3 Overall Evaluation Original Data

To access the data of overall evaluation on V100 and A100, please refer to following Google Sheet:
Overall Evaluation Data

A.4 LightSeq-Hugging Face Guide

This will be a guide on how to use LightSeq in conjunction with Hugging Face, in particular using BERT. This will be a more step-by-step approach rather compared to the section in the Method chapter.

Step 1: Installing LightSeq
Installing LightSeq can be done directly via "pip install LightSeq"

Step 2: Locate the ls_hf_transformer_layer file
This file can either be found in the LightSeq installation folder for Python, or can be copied from our GitHub.

Step 3: Create a model from LSBert
Create the model from LSBert_from_pretrained, like with the Hugging Face pretrained model, using the same parameters as for Hugging Face.

```
model = LSBertForSequenceClassification.from_pretrained(
        model_args.model_name_or_path,
```

```
        training_args=train_args,
        model_args=model_args,
        ls_max_batch_tokens=args.max_batch_tokens,
        config=config,
    )
```

Step 4: Train as normal, using either a custom training loop or using the Hugging Face trainer, just pass the LSBert model as the model.